TMS320C6455 Chip Support Library API Reference Guide

September 2006

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information 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.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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

Preface

Read This First

About This Manual

The API reference guide serves as a software programmer’s handbook for working with the TMS320C6455 CSL.

The purpose of this document is to identify the set of published Chip Support Library (CSL) APIs for the TMS320TCI6455 device. The application developer is expected to refer to this document while designing applications that use these modules.

Abbreviations

Table of Abbreviations

Abbreviation	Description
API	Application Programming Interface
CSL	Chip Support Library
EDMA	Enhanced Direct Memory Access
MCBSP	Multi Channel Serial Processor
VCP	Viterbi Decoder Coprocessor
TCP	Turbo Decoder Coprocessor
TSC	Time Stamp Counter
IDMA	Internal DMA
DDR	Double Data Rate
EMIF	External Memory Interface
GPIO	General Purpose Input/Output
I2C	Inter Integrated Circuit
HPI	Host Port Interface
INTC	Interrupt Controller
PLLC	PLL Controller
SRIO	Serial Rapid IO
BWMNGMT	Bandwidth Management
MEMPROT	Memory Protection
CFG	Configuration
PWRDWN	Power Down

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

Topics

1.1 Introduction

The Chip Support Library layer constitutes a set of well-defined API that abstracts low-level details of the underlying SoC device so that user can configure, control (start/stop etc.) and have read/write access to peripherals without having to worry about register bit-field details.

The CSL services are implemented as distinct modules that correspond with the underlying SoC device modules themselves. By design, CSL APIs follow a consistent style, uniformly across Processor Instruction Set Architecture and are independent of OS. This helps in improving portability of code written using CSL.

1.2 Overview

CSL is realized as twin-layer – a basic register-layer and a more abstracted functional-layer. The lower register layer comprises of a very basic set of macros and type definitions. The upper functional layer comprises of “C” functions that provide an increased degree of abstraction, but intended to provide “directed” control of underlying hardware.

It is important to note that CSL does not manage data-movement over underlying h/w devices. Such functionality is considered a prerogative of a device-driver and serious effort is made to not blur the boundary between device-driver and CSL services in this regard.

CSL does not model the device state machine. However, should there exist a mandatory (hardware dictated) sequence (possibly atomically executed) of register reads/writes to setup the device in chosen “operating modes” as per the device datasheet, then CSL does indeed support services for such operations.

The CSL services are decomposed into modules, each following the twin-layer of abstraction described above. The APIs of each such module are completely orthogonal (one module’s API does not internally call API of another module) and do not allocate memory dynamically from within. This is key to keeping CSL scalable to fit the specific usage scenarios and ease the effort to ROM a CSL based application.

1.3 CSL Interface

CSL is organized into modules by peripheral. Each module contains a twin-layer user interface, the register layer and the functional layer.

The register layer header file for a peripheral <module> is provided in a header file called cslr_<module>.h The functional layer header file for a given peripheral <module> is provided in a header file called csl_<module>.h

In addition to modules for individual peripherals, CSL provides some chip-level modules that perform system and device-level services. These modules are described in the table below.

Table 1: Chip Level Modules

Module	Description
CHIP	Contains the generic device-specific information that is not specific to a peripheral or module. It includes the chip register IDs, field definitions, register read and write functions,
VERSION	Provides for version management, such as chip ID and version ID.
INTC	The interrupt module provides interrupt management services and a dispatcher. This module is delivered as a separate library.

These modules follow the same naming convention for header files.

1.4 Functional Layer

The CSL Functional Layer for TMS320C6455 is provided as a mix of CSL 3.x style and CSL 2.x style recommended application programmer’s interface to the peripheral. To take advantage of hardware abstraction and maintain maximum forward compatibility in the future, users are encouraged to make use of the Functional Layer APIs in their application and driver code.
CSL 3.x supports a core set of Functional Layer APIs across all modules.

Table 2 General format for Functional layer APIs

API	Description
CSL_<mod>Init()	Peripheral initialization function. Optional; does not affect hardware.
CSL_<mod>Open()	Returns a handle to the peripheral instance
CSL_<mod>Close()	Releases handle to peripheral instance
CSL_<mod>HwSetup()	Configures all peripheral registers with values passed in CSL_<mod>HwSetup structure
CSL_<mod>GetHwStatus()	Queries the current peripheral configuration with CSL_<mod>GetHwSetup structure.
CSL_<mod>HwControl()	Performs modification of one or more parameters according to passed command parameter.
CSL_<mod<HwSetupRaw()	Initializes the device registers with the registervalues passed in the Config Data structure.
CSL_<mod>Read()	Data read (I/O peripherals)
CSL_<mod>Write()	Data write (I/O peripherals)

In addition, there may be unique APIs implemented for a peripheral to perform specific operations, as needed. For example “CSL_dmaxGetNextFreeParamEntry” API of DMAX module searches for the next free parameter entry in resource table.

Interface functions exported by this layer are “run to completion”, meaning, they shall not support asynchronous behavior or deferred completion. If the peripheral hardware has ability to initiate a transaction and assert its completion at a later point in time via designated CPU interrupt, the same should be accommodated by higher-level software (typically device drivers). In general, CSL APIs do not perform resource management or memory allocation; this is managed by the application code or device drivers.

1.4.1 CSL Basic Data Types

The following basic data types are defined in CSL

Table 2: CSL Basic Data Types

Type	Defined as.
Bool	**Unsigned short**
Int	**int**
Char	**char**
String	**char ***
Ptr	**void ***
Uint32	**unsigned int**

Uint16	**unsigned short**
Uint8	**unsigned char**
Int32	**int**
Int16	**short**
Int8	**char**
CSL_BitMask8	**Uint8**
CSL_BitMask16	**Uint16**
CSL_BitMask32	**Uint32**
CSL_Reg8	**volatile Uint8**
CSL_Reg16	**volatile Uint16**
CSL_Reg32	**volatile Uint32**
CSL_Status	**Int16**

1.4.2 Functional Layer Naming Conventions

The CSL reserved names fall into two categories, those that are ***declared*** (ex: Functions, variables and so on) and those that are ***symbolic constants*** and ***macros*** that are implemented via enum or #defines. The declarative names should strictly be avoided from redefining by user. The #defines however, are open for redefinition via the standard C supported #undef construct. Regardless, user is encouraged not to redefine/conflict with CSL Namespace, as side effects are hard to predict.

The following table illustrates the CSL naming conventions:

Table 3: CSL 3.x naming conventions

Format	Namespace	Type
CSL_<MODULE>_<STRING>	Symbolic constant specified as either a #define or an enum. The entire name must be in upper case the <MODULE> denotes peripheral module name and the <STRING> denotes any name, representative of the item being specified or defined. The <STRING> part can have one or more underscores embedded for improved readability.	**CSL_INTC_EVENTID_CNT** Here **INTC** is <MODULE>
CSL_<PeriTitleCaseName>	Peripheral module data type. The CSL_ prefix will be in upper case. The module name string is capitalized and follows title case convention without any underscores. Upper case is used to denote start of a new word or phrase.	**CSL_T**imer**O**bj **CSL_I**ntc**E**vent**I**d **CSL_I**2c**H**w**S**etup **CSL_I2cH**andle

Table 4: CSL 2.x naming conventions

Format	Namespace	Type
<MODULE>_<STRING>	Symbolic constant specified as either a #define or an enum.	MCBSP_RCV_START Here MCBSP is

	The entire name must be in upper case the <MODULE> denotes peripheral module name and the <STRING> denotes any name, representative of the item being specified or defined. The <STRING> part can have one or more underscores embedded for improved readability.	<MODULE>
<MODULE>_<TitleCaseName>	Peripheral module data type. The <MODULE> prefix will be in upper case. The name string is capitalized and follows title case convention without any underscores. Upper case is used to denote start of a new word or phrase.	MCBSP_Obj HPI_Config

1.4.3 Symbolic Constants

This section documents the symbolic (#define) constants that constitute part of published CSL APIs. The table only lists the common symbols that are applicable to all peripheral modules. However, there exists a whole host of symbolic constants that are very specific to each particular module and are **not** listed here.

Table 5: Symbolic constants naming conventions

Name	**Description**
CSL_<MODULE>_<n>_REGS	Base address of hardware registers for instance <n> of said peripheral module. Ex: CSL_TIMER_1_REGS for TimeR
CSL_<MODULE>_CHA<m>_REGS	Base address of hardware registers for channel <m> of peripheral, for modules that do support multiple channels or resources.

1.4.4 Error Codes

The CSL3.x will extend minimal support for error handling. Essentially, CSL will only report success or failure of APIs via their return types and/or separate status parameter passed to the call itself.

The error codes are 16bit signed binary numbers that allows us to represent 32 K unique errors. The entire space is divided into 1024 groups, each of size 32. The first group is reserved for CSL generic system errors, the second through last are distributed amongst individual CSL modules. A positive number is regarded as OK status and/or successful operation of a CSL API. All error states are represented as negative integers only.The following table documents the base set of CSL error codes, **not** specific to any given peripheral.

Table 6: Common error codes

Error Code	Number	Description
CSL_SOK	+1	Success
CSL_ESYS_FAIL	-1	Generic failure
CSL_ESYS_INUSE	-2	Peripheral resource is already in use
CSL_ESYS_XIO	-3	Encountered a shared I/O (XIO) pin conflict
CSL_ESYS_OVFL	-4	Encountered CSL system resource overflow
CSL_ESYS_BADHANDLE	-5	Handle passed to CSL was invalid
CSL_ESYS_INVPARAMS	-6	Invalid parameters.
CSL_ESYS_INVCMD	-7	Command passed to the CSL was invalid.
CSL_ESYS_INVQUERY	-8	Query passed to the CSL was invalid.
CSL_ESYS_NOTSUPPORTED	-9	Action not supported by CSL.

1.5 Register Layer

1.5.1 Register Layer Naming Conventions

All names are alphanumeric except for use of underscores as delimiters.

Table 7: Register layer naming conventions

Convention	Description
CSL Module Identifiers	CSL_<MOD>_ID, where <MOD> is name of the CSL module for a specific peripheral Ex: CSL_TIMER_ID, CSL_MCBSP_ID
Peripheral Instance Identifiers	For CSL modules, which have more than one instance, the convention followed is CSL_<MOD>_*<NUM>*, where <NUM> is Instance number as per Device Data Sheet or Peripheral Reference Guide Ex: CSL_TIMER_1, CSL_MCBSP_1, CSL_I2C_1 For CSL modules, which have single instance, the convention followed is CSL_<MOD> Ex: CSL_EDMA3, CSL_GPIO
Peripheral Instance Count Identifiers	CSL_<MOD>_CNT, where <MOD> is peripheral module whose number of instances is defined Ex: CSL_EMIFA_CNT, CSL_HPI_CNT

Peripheral Register Identifiers	<MOD>_<REG>, where <REG> is register name. Names used with specific peripheral instance overlays. Ex: TIMER_CNTL, CPMAC_TX_CONTROL, CPMAC_RX_CONTROL
Peripheral Register Continuous Bit-field Identifiers	<MOD>_<REG>_<FIELD>, where <FIELD> is bit-field name. Names are instance-dependent. Ex: TIMER_CNTL_CLKEN, CPMAC_TX_CONTROL_EN
Peripheral Register Bit-field Symbols	<MOD>_<REG>_<FIELD>_<SYM>, where <SYM> is bit-field setting symbolic token Names are instance-dependent. Ex: CPMAC_TX_CONTROL_EN_RESETVAL, TIMER_CNTL_CLKEN_SHIFT

1.5.2 Register Overlay Structure

SoC peripherals are typically programmed by reading/writing to one or more registers in the peripheral's IO address space. In order to allow for clean and intuitive access to all the registers belonging to a given peripheral instance, CSL implements a technique called Register Overlay Structure. A C data structure template is defined with structure members corresponding to each of the registers of the peripheral device in the order in which they occur. The member types are chosen to correspond to the widths of the register they represent. Appropriate padding is introduced to ensure alignment for proper addressing of these registers from "C". The structure members use names that correspond to those used in the peripheral datasheet, to ease programming. Since there exists a well-formed C structure, the registers can be viewed in IDE watch windows and presumably recognized by smart-editors that can do auto-completion while typing.

It should be noted that register overlays do not consume memory, as they are not instantiated. The purpose of these structures is mainly to typify the "C" pointers

Example:

The figure below shows the layout of a TIMER peripheral device. Assuming there exist two instances of the device, one at address 0x02940000 and the other at address 0x02980000, the register overlay for such a device is specified as follows:

```
typedef struct {
    /* Timer Control Register */
    Uint32 CTL;
    /* Timer Period Register */
    Uint32 PRD;
    /* Timer Counter Register */
    Uint32 CNT;
```

```
} CSL_TimerRegs;
typedef volatile CSL_TimerReg *CSL_TimerRegsOvly;
```

Figure 1: Register Layer overlay structure

CSL_TIMER_0_REGS **(0x02940000)**	**CTL [+0x00]**
	PRD [+0x04]
	CNT [+0x08]
CSL_TIMER_1_REGS **(0x02980000)**	**CTL [+0x00]**
	PRD [+0x04]
	CNT [+0x08]

1.5.3 Register Layer Symbolic Constants

The CSL register layer file for a given peripheral device (cslr_<module>.h) will define certain standard symbols for each peripheral register/bit field. These symbolic constants are declared with the following convention:

Notational convention: CSL_<MODULE>_<REG>_<FIELD>_<SYMBOL>

The semantics of the various parts of the symbolic name is shown in table below:

Table 8: Symbolic names used in Register layer

Convention	Description
<**MODULE**>	The CSL Peripheral module Ex: **TIMER**
<**REG**>	The peripheral device register Ex: **CNT**
<**FIELD**>	Bit-field of interest Ex: **ST**
<**SYMBOL**>	Operational symbol for constant being defined Ex: **STOP, START**

Ex: CSL_TIMER_CNT_ST_STOP

The table below summarizes the standard symbols used with register bit fields:

Table 9: Standard Symbols used with register bit fields

#define Symbolic Constant	Semantics of the Value Assigned
CSL_<MODULE>_<REG>_SHIFT	The number of left shift positions to reach the register bit-field of interest
CSL_<MODULE>_<REG>_MASK	The binary *and* mask useful to extract register bit-field of interest
CSL_<MODULE>_<REG>_RESETVAL	The power-on reset value assumed by the register or bit-field of interest

NOTE: The above defines specified in **cslr_<module>.h** have math bit ordering of MSB: LSB and are regardless of what Endian-flips occur as these are read over processor memory busses. Typically, the processor hardware wiring will be such that CPU always gets to read/write its memory mapped peripherals in "Native Endian" format i.e., ready for CPU interpretation. Should there be Endian mismatch between CPU and memory-mapped peripheral, then necessary corrections (Swaps) must be handled outside, before applying the **_MASK**, **_SHIFT** etc., symbols shown in this file.

1.5.4 Register Layer Macros

Table 10: Register Bit Field Manipulation Macro services

Service	Description
CSL_FMK(field, val)	Creates an AND mask of value (val) moved to specified field location.
CSL_FMKT (field, token)	Same as CSL_FMK, but allows predefined symbolic tokens to be used as value.
CSL_FMKR (msb, lsb, val)	Same as CSL_FMK, but allows raw bit positions (msb:lsb) to specify bit-field.
CSL_FEXT(reg, field)	Evaluates the arithmetic value of bits gathered from specified field.
CSL_FEXTR(reg, msb, lsb)	Same as CSL_FEXT, but allows raw bit positions (msb:lsb) to specify bit-field.
CSL_FINS(reg, field, val)	Inserts the specified value (val) at the specified field in the register.
CSL_FINST(reg, field, token)	Same as CSL_FINS, but allows predefined symbolic tokens to be used as value.
CSL_FINSR(reg, msb, lsb, val)	Same as CSL_FINS, but allows raw bit positions (msb:lsb) to specify bit-field.

1.6 C++ Compatibility

CSL Functional Layer APIs are, for the most part, implemented in C, with small parts implemented in native assembly to work around some difficulties of realizing the same in C. Regardless, the APIs are declared appropriately so as to allow C++ applications to call them. Unlike C++ functions, the CSL APIs will not support specification of default values for formal arguments passed to them.

Also, in places where CSL API semantics require the user to specify function pointers, CSL3.x design does not allow the user to input a C++ function pointer. To work around above limitation, a wrapper function in C, encapsulating the C++ member function needs to be written by the user. This function can be designed to input the class instance as an argument (along with any other parameters that it requires), and invoke the appropriate class member function internally for achieving the desired objectives.

1.7 INTC Software Architecture

The INTC module in CSL3.x is designed to provide an abstraction for all the basic interrupt controller functions, such as enabling and disabling interrupts, specifying the user function to be called in response to interrupts, and setting desired hardware properties. These functions are not specific to any given peripheral. In other words, the INTC module abstracts the generic interrupt capabilities supported by the processor.

NOTE: The CSL 3.0 INTC module is delivered as a separate library from the remaining CSL modules. When using an embedded operating system that contains interrupt controller/dispatcher support, do not link in the INTC library. For interrupt controller support, DSP/BIOS users should use the HWI (Hardware Interrupt) and ECM (Event Combiner Manager) modules supported under DSP/BIOS v5.21 or later.

This section describes the CSL INTC functionality for C6482.

1.7.1 The Interrupt Controller

The figure below shows a multi-level interrupt controller, within the dashed line boundary. The Level-0 controller is considered part of the CPU itself. This level implements the primary Interrupt Vector Table for the processor. The remaining controllers help expand these vectors to handle many more hardware events. These events, shown as arrows, might be externally triggered via device input pins and/or internally asserted by on-chip peripheral devices. The peripheral devices themselves, represented by cross-hatched boxes, might host an event controller (checkered box) that helps map the hardware events to outgoing lines that assert the CPU interrupts. Each of the interrupt controllers would have programmable control registers to enable/disable the hardware events from proceeding towards the CPU Interrupts. The controllers also allow the user to configure the interrupt capture circuitry to a specified polarity, edge sensitivity, priority, etc. The Level-0 interrupt controller is shown as having a “selector” capability that maps a given CPU interrupt vector to one-of-N input hardware events. This scheme, although available at Level-0 in today’s TI processors, is technically possible to be also present at other levels. It is also possible for an interrupt controller at a given level to be comprised of more than one logical block. (See 2.0, 2.1 blocks in the figure.) All of the blocks that comprise the interrupt controllers Level-0 through level-N are part of the INTC module (bounded by dashed line) in CSL3.x. Only the top level INTC abstraction will be seen by the user. The internal INTC0 through INTCn are hidden from user. The wiring between individual INTC sub-blocks shall be done during INTC module initialization and dispatcher setup as detailed in ensuing sub-sections of this document. It is important to note that any “custom controllers” (refer to checkered box in figure) embedded in specific peripheral devices (ex: C64x EDMA Controller) are not considered part of INTC functionality.

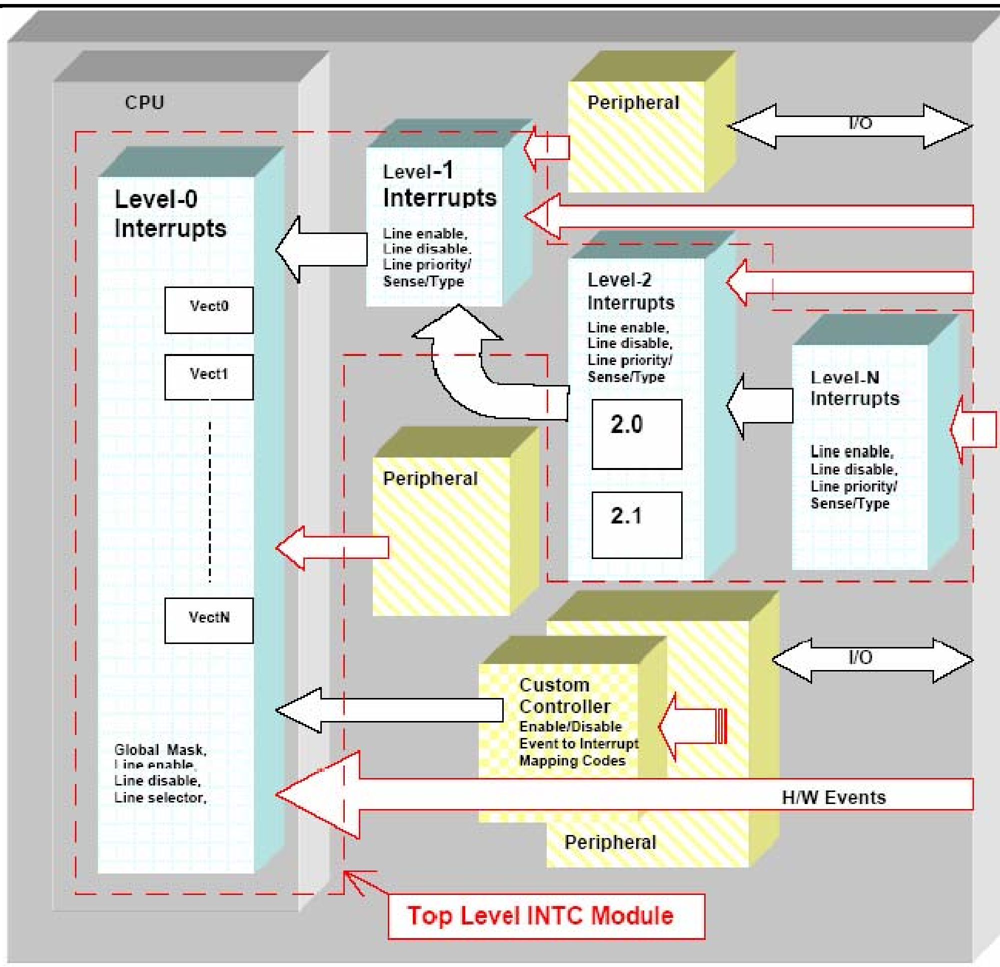

Figure 2 INTC Controller block diagram

1.7.2 INTC Module Initialization

The INTC module maintains an array of bit masks that enable INTC to keep track of interrupts that are active or in-use by the application. Each bit position corresponds to a single hardware event that can be processed by the INTC. The total bit positions maintained corresponds to the maximum number of hardware events that the INTC can recognize and handle at any given time. At level-0, this corresponds to the CPU primary interrupt vectors; at other levels, it corresponds to the capacity of fan-in and priority resolution implemented in the controllers. The INTC module will assign unique IDs to each hardware event and has knowledge of the range of such IDs applicable at each level (ie., INTC0 thru' INTCn).

During the CSL initialization phase, the INTC module initialization function CSL_intcInit() is called. This will reset the global variable CSL_IntcContext.eventAllocMask[] to zeros. This implies that all the interrupts are available for use by the application. This array stands responsible for resolving any conflicts on the same interrupt resource. Only one interrupt service routine can be plugged in for each interrupt. Thus when the same interrupt line is shared between different events, synchronization of the events has to be taken care in the application program and the event that caused the interrupt has to be identified in the ISR to get the proper function executed.

1.7.3 Interrupt Dispatcher Specifics

Following successful initialization of INTC module (via CSL_intcInit()), the user can choose to nitialize the INTC built-in dispatcher by calling CSL_intcDispatcherInit(). The dispatcher record argument passed for CSL_intcDispatcherInit()) is used as a record of which ISR is hooked to a particular CPU interrupt at a particular point in time. Once the dispatcher record is created and initialized, CSL_plugEventHandler() will internally perform recording the ISR in the dispatcher record and hooking up the appropriate primary ISRs in the interrupt vector table.

Typically, when an operating system (OS) is running, the primary interrupt vector table is under control of the OS Scheduler. The OS Scheduler will hook its own dispatcher function at this level-0. OS ports can choose to either do away completely with CSL dispatchers or implement their own for the desired levels. They can choose to first initialize the CSL interrupt dispatcher and then swap-in their own interrupt handlers at desired levels and/or vectoring slots. When an OS port used with CSL will use its own dispatcher, the CSL_IntcContext.flags must be equal to CSL_INTC_CONTEXT_DISABLECOREVECTORWRITES. The CSL dispatch code/data may either not be loaded at all, or be overlaid for optimizing on memory footprint. Typically, the event dispatch record constitutes the context information. This table will hold the address of the event handler function and pointer to arbitrary data object (void*) to be passed to event handler as a lone argument.

When INTC Dispatcher will not be used, the CSL_intcHookIsr() can be used to hook the right fetch packet in the Interrupt Vector Table, which in turn leads the CPU control to the right ISR on occurrence of the interrupt.

1.7.4 INTC API Call Sequence

The sequence of calls made by INTC user will be as follows –

```
// Initialize other required CSL3.x Peripherals
CSL_intcSetVectorPtr(DEST_ADDR); //If relocation of interrupt vector
                                 //required. DEST_ADDR is new location
CSL_intcInit();                  // INTC Module Initialize
CSL_intcDispatcherInit();        // Dispatcher Initialize (if reqd.)
CSL_intcGlobalDisable(..);       // Disable global interrupts.
handle = CSL_intcOpen(..);       // Ready an interrupt for use
CSL_intcHwSetup(handle..);       // Setup interrupt attributes
CSL_intcPlugEventHandler(..);    // Bind the interrupt with the
                                 // corresponding ISR.
CSL_intcHwControl(handle..);     // assorted control, ex: ISR hookup
CSL_intcEventEnable(..);         // Enables the event of interest.
CSL_intcGlobalEnable(..);        // Enable global interrupts.
:
CSL_intcClose(handle);           // End of interrupt use
                                 // Terminate Program
```

Chapter 2
DAT MODULE

Topics

2.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within DAT module.

The data module (DAT) is used to move data around by means of EDMA hardware. This module serves as a level of abstraction such that it works the same for devices that have the EDMA peripheral as for devices that have the EDMA peripheral.

Unlike the previous DAT CSL, the resources required by the DAT should be managed by the user. The user must ensure that the TCC number, QDMA channel provided as an argument to DAT_open () are not used by any other EDMA applications. DAT CSL doesnot support multiple transfer requests pending. The second transfer is submitted only after the completion of the first transfer. Since only a single transfer can be outstanding in the DAT, correlation between the transfer submitted and the completion code is done by TCC number, no unique transfer ID is maintained.

2.2 Functions

This section lists the functions available in the DAT module.

2.2.1 DAT_open

ICSL_Status DAT_open ([DAT_Setup](#) * *setup*)

Description
This API,

a. Sets up the channel to Parameter set mapping
b. Sets up the priority. This is essentially done by specifying the queue to which the channel is submitted to viz. Queue0- Queue3 with Queue 0 being the highest priority.
c. Enables the region access bit for the channel, if a region is specified.

Arguments

```
setup          Pointer to the DAT setup structure
```

Return Value
CSL_Status

```
CSL_SOK -
```

Pre Condition
None

Post Condition
The EDMA registers are configured with the setup values passed.

Modifies
EDMA registers

Example

```
DAT_Setup datSetup;
CSL_Status status;
datSetup.qchNum     = CSL_DAT_QCHA_0;
datSetup.regionNum  = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum     = 1;
datSetup.paramNum   = 0 ;
datSetup.priority   = CSL_DAT_PRI_0;

status = DAT_open(&datSetup);
...
```

2.2.2 DAT_close

void DAT_close (void)

Description
This API disables the region access bit, if specified.

Arguments
None

Return Value
None

Pre Condition
DAT_open() must be successfully invoked prior to this call.

Post Condition
None

Modifies
None

Example

```
DAT_Setup datSetup;
datSetup.qchNum     = CSL_DAT_QCHA_0;
datSetup.regionNum  = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum     = 1;
datSetup.paramNum   = 0 ;
datSetup.priority   = CSL_DAT_PRI_0;

DAT_open(&datSetup);
...
DAT_close();
...
```

2.2.3 DAT_copy

Uint32 DAT_copy	**(**	**void ***	***src*,**
		void *	***dst*,**
		Uint16	***byteCnt***
	)		

Description
This API copies a linear block of data from `src` to `dst` using EDMA hardware, depending on the device. The arguments are checked for alignment. For best efficiency, the source and destination addresses should be aligned on an 8-byte boundary, with the transfer rate a multiple of eight.

Arguments

```
src         Source memory address for the data transfer

dst         Destination memory address of the data transfer

byteCnt     Number of bytes to be transferred
```

Return Value
`Uint32`
tccNum - Transfer completion code

Pre Condition

DAT_open() must be successfully invoked prior to this call.

Post Condition
The EDMA registers are configured to transfer byteCnt bytes from the source memory address to the destination memory address.

Modifies
EDMA registers

Example

```
DAT_Setup     datSetup;
Uint8         dst1d[8*16];
Uint8         src1d[8*16];
Uint32        tccNum;
datSetup.qchNum     = CSL_DAT_QCHA_0;
datSetup.regionNum  = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum     = 1;
datSetup.paramNum   = 0 ;
datSetup.priority   = CSL_DAT_PRI_0;

DAT_open(&datSetup);
tccNum = DAT_copy(&src1d,&dst1d,256);
DAT_close();
...
```

2.2.4 DAT_fill

Uint32 DAT_fill	**(**	**void ***	***dst*,**
		Uint16	***byteCnt*,**
		Uint32 *	***value***
	)		

Description
This API fills a linear block of memory with the specified fill value using EDMA hardware

Arguments

```
dst          Destination memory address to be filled

byteCnt      Number of bytes to be filled

value        Value to be filled
```

Return Value
`Uint32`

tccNum - Transfer completion code

Pre Condition
DAT_open() must be successfully invoked prior to this call.

Post Condition

The EDMA registers are configured to transfer a value to byteCnt bytes of the destination memory address.

Modifies
EDMA registers

Example

```
DAT_Setup     datSetup;
Uint8         dst[8*16];
Uint32        fillVal;
Uint32        tccNum;

datSetup.qchNum      = CSL_DAT_QCHA_0;
datSetup.regionNum   = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum      = 1;
datSetup.paramNum    = 0 ;
datSetup.priority    = CSL_DAT_PRI_0;
DAT_open(&datSetup);
...
fillVal              = 0x5a;
tccNum = DAT_fill(&dst,256,&fillVal);
...
DAT_close();
...
```

2.2.5 DAT_wait

void DAT_wait (Uint32 *id*)

Description
This API Waits for completion of the ongoing transfer.

Arguments

```
id         Transfer completion number of the previous transfer
```

Return Value
None

Pre Condition
DAT_copy()/DAT_fill must be successfully invoked prior to this call.

Post Condition
Indicates that the transfer ongoing is complete.

Modifies
None

Example

```
DAT_Setup     datSetup;
Uint8         dst1d[8*16];
Uint8         src1d[8*16];
Int           id;
```

```
datSetup.qchNum = CSL_DAT_QCHA_0;
datSetup.regionNum = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum = 1;
datSetup.paramNum = 0 ;
datSetup.priority = CSL_DAT_PRI_0;

DAT_open(&datSetup);
...
id = DAT_copy(&src1d,&dst1d,256);

DAT_wait(id);
...
DAT_close();
...
```

2.2.6 DAT_busy

Int16 DAT_busy (Uint32 *id*)

Description
This API polls for transfer completion.

Arguments

```
id          Transfer completion number of the previous transfer
```

Return Value
`Int16`

- TRUE/FALSE

Pre Condition
DAT_copy()/DAT_fill must be successfully invoked prior to this call.

Post Condition
Indicates that the transfer ongoing is complete.

Modifies
None

Example

```
DAT_Setup     datSetup;
Uint8         dst1d[8*16];
Uint8         src1d[8*16];
Int           id;

datSetup.qchNum = CSL_DAT_QCHA_0;
datSetup.regionNum = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum = 1;
datSetup.paramNum = 0 ;
datSetup.priority = CSL_DAT_PRI_0;

DAT_open(&datSetup);
...
id = DAT_copy(&src1d,&dst1d,256);
```

```
do {
    ...
} while (DAT_busy(id));
...
DAT_close();
...
```

2.2.7 DAT_copy2d

Uint32 DAT_copy2d	**(**	**Uint32**	***type*,**
		void *	***src*,**
		void *	***dst*,**
		Uint16	***lineLen*,**
		Uint16	***lineCnt*,**
		Uint16	***linePitch***
	)		

Description
This API copies data from source to destination for two dimension transfer.

Arguments

```
type        Indicates the type of the transfer
            DAT_1D2D - 1 dimension to 2 dimension
            DAT_2D1D - 2 dimension to 1 dimension
            DAT_2D2D - 2 dimension to 2 dimension

src         Source memory address for the data transfer

dst         Destination memory address of the data transfer

lineLen     Number of bytes per line

lineCnt     Number of lines

linePitch   Number of bytes between start of one line to start
            of next line
```

Return Value
`Uint32`
TccNum – Transfer completion code

Pre Condition
DAT_open() must be successfully invoked prior to this call.

Post Condition
The EDMA registers are configured for the transfer.

Modifies
EDMA registers

Example

```
DAT_Setup     datSetup;
Uint8         dst2d[8][20];
Uint8         src1d[8*16];
Int           id;

datSetup.qchNum = CSL_DAT_QCHA_0;
datSetup.regionNum = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum = 1;
datSetup.paramNum = 0 ;
datSetup.priority = CSL_DAT_PRI_0;

DAT_open(&datSetup);
...
id = DAT_copy2d(DAT_1D2D,src1d,dst2d,16,8,20);

do {
...
}while (DAT_busy(id));
...
DAT_close();
...
```

2.2.8 DAT_setPriority

void DAT_setPriority (Int *priority*)

Description
Sets the priority bit value PRI of OPT register. The priority value can be set by using the type CSL_DatPriority.

Arguments

```
priority          Priority value
```

Return Value
None

Pre Condition
DAT_open must be successfully invoked prior to this call.

Post Condition
OPT register is set for the priority value

Modifies
OPT register

Example

```
DAT_Setup     datSetup;
Uint8         dst2d[8][20];
Uint8         src1d[8*16];
datSetup.qchNum = CSL_DAT_QCHA_0;
datSetup.regionNum = CSL_DAT_REGION_GLOBAL ;
datSetup.tccNum = 1;
```

```
datSetup.paramNum = 0 ;
datSetup.priority = CSL_DAT_PRI_0;

DAT_open(&datSetup);
...
DAT_setPriority(CSL_DAT_PRI_3);
...
```

2.3 Data Structures

This section lists the data structures available in the DAT module.

2.3.1 DAT_Setup

Detailed description
DAT Setup structure.

Field Documentation

Int DAT_Setup::paramNum
Parameter set number for this channel

Int DAT_Setup::priority
Priority/Queue number on which the transfer requests are submitted

Int DAT_Setup::qchNum
QDMA Channel number being requested

Int DAT_Setup::regionNum
Region of operation

Int DAT_Setup::tccNum
Transfer completion code dedicated for DAT

2.4 Macros

#define DAT_1D2D 0x1
Transfer type is 1D2D

#define DAT_2D1D 0x2
Transfer type is 2D1D

#define DAT_2D2D 0x3
Transfer type is 2D2D

Chapter 3
DDR2 Module

Topics

3.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within DDR2 module.

This is a 32-bit DDR2 SDRAM interface. The 32-bit DDR2 Memory Controller bus is used to interface to DDR2 devices. The DDR2 external bus only interfaces to DDR2 devices; it does not share the bus with any other types of peripherals. The DDR2 memory controller interfaces with JESD79D-2A standard compliant DDR2 SDRAM devices. Memory types such as DDR1 SDRAM, SDR SDRAM, SBSRAM, and asynchronous memories are not supported. The DDR2 memory controller SDRAM can be used for program and data storage.

The DDR2 memory controller supports the following features:

- JESD79D-2A standard compliant DDR2 SDRAM
- 256 Mbyte memory space
- Data bus width of 32 or 16 bits
- CAS latencies: 2, 3, 4, and 5
- Internal banks: 1, 2, 4, and 8
- Burst length: 8
- Burst type: sequential
- 1 CE signal
- Page sizes: 256, 512, 1024, and 2048
- SDRAM autoinitialization
- Self-refresh mode
- Prioritized refresh
- Programmable refresh rate and backlog counter
- Programmable timing parameters
- Little-endian and big endian transfers

3.2 Functions

This section lists the functions available in the DDR2 module.

3.2.1 CSL_ddr2Init

CSL_Status CSL_ddr2Init (CSL_Ddr2Context * *pContext*)

Description
This is the initialization function for the DDR2 CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context. As DDR2 doesn't have any
            context based information user is expected to pass
            NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
The CSL for DDR2 is initialized.

Modifies
None

Example

```
...
if (CSL_SOK != CSL_ddr2Init(NULL)) {
    return;
}
...
```

3.2.2 CSL_ddr2Open

CSL_Ddr2Handle CSL_ddr2Open (CSL_Ddr2Obj * *pDdr2Obj*, CSL_InstNum *ddr2Num*, CSL_Ddr2Param * *pDdr2Param*, CSL_Status * *pStatus*)

Description
This function returns the handle to the DDR2 instance. The open call sets up the data structures

for the particular instance of DDR2. The handle returned by this call is input argument for rest of the DDR2 CSL APIs.

Arguments

```
pDdr2Obj        Pointer to the object that holds reference to the
                instance of DDR2 requested after the call

ddr2Num         Instance of DDR2 to which a handle is requested

pDdr2Param      Pointer to module specific parameters

pStatus         pointer for returning status of the function call
```

Return Value
`CSL_Ddr2Handle`

- Valid DDR2 instance handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The DDR2 must be successfully initialized via CSL_*ddr2Init*() before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is
- CSL_SOK - Valid DDR2 handle is returned.
- CSL_ESYS_FAIL - The DDR2 instance is invalid.
- `CSL_ESYS_INVPARAMS` – The object passed is invalid.

2. DDR2 object structure is populated.

Modifies
1. The status variable
2. object structure

Example:
```
CSL_Status          status;
CSL_Ddr2Obj         ddr2Obj;
CSL_Ddr2Handle      hDdr2;
...
hDdr2 = CSL_Ddr2Open(&ddr2Obj,
                     CSL_DDR2,
                     NULL,
                     &status);
...
```

3.2.3 CSL_ddr2Close

CSL_Status CSL_ddr2Close (CSL_Ddr2Handle *hDdr2*)

Description
This function closes the specified instance of DDR2.

Arguments
```
hDdr2          DDR2 handle returned by successful 'open'
```

Return Value
`CSL_Status`

- `CSL_SOK` - DDR2 close successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Both *CSL_ddr2Init()* and *CSL_ddr2Open()* must be called successfully in order before calling *CSL_ddr2Close()*.

Post Condition
The DDR2 CSL APIs can not be called until the DDR2 CSL is reopened again using *CSL_ddr2Open()*.

Modifies
Obj structure values

Example

```
CSL_Ddr2Handle    hDdr2;
CSL_Status        status;
...
status = CSL_ddr2Close(hDdr2);
...
```

3.2.4 CSL_ddr2HwSetup

CSL_Status CSL_ddr2HwSetup (CSL_Ddr2Handle *hDdr2*, CSL_Ddr2HwSetup * *setup*)

Description
This function initializes the device registers with the appropriate values provided through the HwSetup data structure. For information passed through the HwSetup data structure, refer CSL_ddr2HwSetup.

Arguments

```
hDdr2          DDR2 handle returned by successful 'open'

setup          Pointer to setup structure, which contains the
               information to program DDR2 to a required state
```

Return Value
`CSL_Status`

- `CSL_SOK` – Hwsetup successful
- `CSL_ESYS_BADHANDLE` – Handle passed is invalid
- `CSL_ESYS_INVPARAMS` – The param passed is invalid

Pre Condition

Both *CSL_ddr2Init()* and *CSL_ddr2Open()* must be called successfully in order before this function.

Post Condition
DDR2 registers are configured according to the hardware setup parameters.

Modifies
DDR2 registers

Example:

```
CSL_Ddr2Handle hDdr2;
CSL_Status status;
CSL_Ddr2Timing1 tim1 = CSL_DDR2_TIMING1_DEFAULTS;
CSL_Ddr2Timing2 tim2 = CSL_DDR2_TIMING2_DEFAULTS;
CSL_Ddr2Settings set = CSL_DDR2_SETTING_DEFAULTS;
CSL_Ddr2HwSetup hwSetup;

hwSetup.refreshRate      = (Uint16)0x753;
hwSetup.timing1Param     = &tim1;
hwSetup.timing2Param     = &tim2;
hwSetup.setParam         = &set;
...
status = CSL_ddr2HwSetup(hDdr2, &hwSetup);
...
```

3.2.5 CSL_ddr2GetHwSetup

CSL_Status CSL_ddr2GetHwSetup **(** **CSL_Ddr2Handle** ***hDdr2*****,**
CSL_Ddr2HwSetup * ***setup***
)

Description
This function gets the current setup of the DDR2. The status is returned through *CSL_Ddr2HwSetup*. The obtaining of status is the reverse operation of *CSL_ddr2HwSetup()* function.

Arguments

```
hDdr2         DDR2 handle returned by successful 'open'

setup         Pointer to the hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Get Hardware setup successful
- `CSL_ESYS_INVPARAMS` - Param passed is invalid
- `CSL_ESYS_BADHANDLE` - Handle is not valid

Pre Condition
Both *CSL_ddr2Init()* and *CSL_ddr2Open()* must be called successfully in order before calling *CSL_ddr2GetHwSetup()*.

Post Condition
None

Modifies
Second parameter setup value

Example:
```
CSL_Ddr2Handle   hDdr2;
CSL_Status       status;
CSL_Ddr2Timing1  tim1;
CSL_Ddr2Timing2  tim2;
CSL_Ddr2Settings set;
CSL_Ddr2HwSetup  hwSetup;

hwSetup.timing1Param    = &tim1;
hwSetup.timing2Param    = &tim2;
hwSetup.setParam        = &set;
...
status = CSL_ddr2GetHwSetup(hDdr2, &hwSetup);
...
```

3.2.6 CSL_ddr2HwControl

CSL_Status CSL_ddr2HwControl	**(**	CSL_Ddr2Handle	***hDdr2*,**
		CSL_Ddr2HwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
Control operations for the DDR2. For a particular control operation, the pointer to the corresponding data type needs to be passed as argument HwControl function Call. All the arguments (structure elements included) passed to the HwControl function are inputs. For the list of commands supported and argument type that can be *void** casted and passed with a particular command refer to *CSL_Ddr2HwControlCmd*.

Arguments

```
hDdr2        DDR2 handle returned by successful 'open'

cmd          The command to this API indicates the action to be
             taken

arg          Optional argument as per the control command
```

Return Value
`CSL_Status`

- `CSL_SOK` - Command successful
- `CSL_ESYS_BADHANDLE` - Handle passed is invalid
- `CSL_ESYS_INVCMD` - Command passed is invalid

Pre Condition
Both *CSL_ddr2Init()* and *CSL_ddr2Open()* must be called successfully in order before calling *CSL_ddr2HwControl()*.

Post Condition
DDR2 registers are configured according to the command passed.

Modifies
DDR2 registers

Example:
```
CSL_Ddr2Handle          hDdr2;
CSL_Status              status;
CSL_Ddr2SelfRefresh     arg;
arg = CSL_DDR2_SELF_REFRESH_DISABLE;
...
status = CSL_ddr2HwControl(hDdr2,
                           CSL_DDR2_CMD_SELF_REFRESH,&arg);
...
```

3.2.7 CSL_ddr2GetHwStatus

CSL_Status CSL_ddr2GetHwStatus	**(**	**CSL_Ddr2Handle**	***hDdr2*,**
		CSL_Ddr2HwStatusQuery	***query*,**
		void *	***response***
	)		

Description
This function is used to read the current device configuration, status flags and the associated registers. For details about the various status queries supported and the associated data structure to record the response, refer to *CSL_Ddr2HwStatusQuery.*

Arguments

`hDdr2`	`DDR2 handle returned by successful 'open'`
`query`	`The query to this API, which indicates the status to be returned`
`response`	`Response from the query.`

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware status call is successful
- `CSL_ESYS_BADHANDLE` - Not a valid Handle
- `CSL_ESYS_INVQUERY` - Invalid Query
- `CSL_ESYS_INVPARAMS` – Parameter response is not properly initialized

Pre Condition
Both *CSL_ddr2Init()* and *CSL_ddr2Open()* must be called successfully in order before calling *CSL_ddr2GetHwStatus()*.

Post Condition
None

Modifies
Third parameter, response value

Example:

```
CSL_Ddr2Handle    hDdr2;
CSL_Status        status;
Uint16            response;
...
status = CSL ddr2GetHwStatus(hDdr2,CSL DDR2 QUERY REFRESH RATE,
                                &response);
...
```

3.2.8 CSL_ddr2HwSetupRaw

CSL_Status CSL_ddr2HwSetupRaw (CSL_Ddr2Handle ***hDdr2***, CSL_Ddr2Config * ***config***)

Description
This function initializes the device registers with the register-values provided through the Config data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hDdr2       Handle to the DDR2 external memory interface instance

config      Pointer to the config structure containing the
            device register values
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration structure pointer is not properly initialized

Pre Condition
CSL_ddr2Init() and *CSL_ddr2Open* () must be called successfully before calling this function.

Post Condition
The registers of the specified DDR2 instance will be setup according to the values passed through the Config structure.

Modifies
Hardware registers of the DDR2

Example

```
CSL_Ddr2Handle    hDdr2;
CSL_Ddr2Config    config = CSL_DDR2_CONFIG_DEFAULTS;
CSL_Status        status;
...
status = CSL_ddr2HwSetupRaw(hDdr2, &config);
...
```

3.2.9 CSL_ddr2GetBaseAddress

CSL_Status CSL_ddr2GetBaseAddress (CSL_InstNum ***ddr2Num*****,**
CSL_Ddr2Param * ***pDdr2Param*****,**
CSL_Ddr2BaseAddress * ***pBaseAddress***
)

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_ddr2Open() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

`ddr2Num`	`Specifies the instance of the DDR2 external memory interface for which the base address is requested`
`pDdr2Param`	`Module specific parameters.`
`pBaseAddress`	`Pointer to the base address structure to return the base address details.`

Return Value
`CSL_Status`

- `CSL_SOK` - Successful on getting the base address of DDR2
- `CSL_ESYS_FAIL` - The DDR2 external memory interface instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure

Example
```
CSL_Status          status;
```

```
CSL_Ddr2BaseAddress  baseAddress;
...
status = CSL_ddr2GetBaseAddress(CSL_DDR2, NULL, &baseAddress);
...
```

3.3 Data Structures
This section lists the data structures available in the DDR2 module.

3.3.1 CSL_Ddr2Obj
Detailed Description
This object contains the reference to the instance of DDR2 opened using the *CSL_ddr2Open()*. The pointer to this is passed to all DDR2 CSL APIs. CSL_ddr2Open() function initializes this structure based on the parameters passed.

Field Documentation

CSL_InstNum CSL_Ddr2Obj::perNum
This is the instance of DDR2 being refered to by this object

CSL_Ddr2RegsOvly CSL_Ddr2Obj::regs
Pointer to the register overlay structure of the DDR2

3.3.2 CSL_Ddr2Config
Detailed Description
DDR2 config structure, which is used in *CSL_ddr2HwSetupRaw()* function. This is a structure of register values, rather than a structure of register field values like CSL_Ddr2HwSetup.

Field Documentation

Volatile Uint32 CSL_Ddr2Config::SDCFG
SDRAM Config Register

Volatile Uint32 CSL_Ddr2Config::SDRFC
SDRAM Refresh Control Register

Volatile Uint32 CSL_Ddr2Config::SDTIM1
SDRAM Timing1 Register

Volatile Uint32 CSL_Ddr2Config::SDTIM2
SDRAM Timing2 Register

Volatile Uint32 CSL_Ddr2Config::BPRIO
VBUSM Burst Priority Register

3.3.3 CSL_Ddr2Context
Detailed Description
DDR2 specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_Ddr2Context::contextInfo
Context information of DDR2 external memory interface CSL passed as an argument to CSL_ddr2Init(). Present implementation of DDR2 CSL doesn't have any context information; hence assigned NULL. The declaration is just a placeholder for future implementation.

3.3.4 CSL_Ddr2Param

Detailed Description
This is module specific parameter. Present implementation of DDR2 CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_Ddr2Param::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation. Passed as an argument to CSL_ddr2Open().

3.3.5 CSL_Ddr2HwSetup

Detailed Description
This has all the fields required to configure DDR2 at Power Up (after a Hardware Reset) or a Soft Reset. This structure is used to setup or obtain existing setup of DDR2 using *CSL_ddr2HwSetup()* and *CSL_ddr2GetHwSetup()* functions respectively.

Field Documentation

Uint16 CSL_Ddr2HwSetup::refreshRate
Refresh Rate

CSL_Ddr2Settings* CSL_Ddr2HwSetup::setParam
Structure for DDR2 SDRAM configuration parameter

CSL_Ddr2Timing1* CSL_Ddr2HwSetup::timing1Param
Structure for DDR2 SDRAM Timing1

CSL_Ddr2Timing2* CSL_Ddr2HwSetup::timing2Param
Structure for DDR2 SDRAM Timing2

3.3.6 CSL_Ddr2BaseAddress

Detailed Description
This structure contains the base address information for the DDR2 instance.

Field Documentation

CSL_Ddr2RegsOvly CSL_Ddr2BaseAddress::regs
Base address of the configuration registers of the peripheral

3.3.7 CSL_Ddr2Timing1

Detailed Description
Timing1 structure to set the Timing1 register of DDR2 SDRAM.

Field Documentation

Uint8 CSL_Ddr2Timing1::tras
Specifies TRAS value: Minimum number of DDR2 EMIF cycles from Activate to Pre-charge command, minus one

Uint8 CSL_Ddr2Timing1::trc
Specifies TRC value: Minimum number of DDR2 EMIF cycles from Activate command to Activate command, minus one

Uint8 CSL_Ddr2Timing1::trcd
Specifies TRCD value: Minimum number of DDR2 EMIF cycles from Active to Read or Write command, minus one

Uint8 CSL_Ddr2Timing1::trfc
Specifies TRFC value: Minimum number of DDR2 EMIF cycles from Refresh or Load command to Refresh or Activate command, minus one

Uint8 CSL_Ddr2Timing1::trp
Specifies TRP value: Minimum number of DDR2 EMIF cycles from Pre-charge to Active or Refresh command, minus one

Uint8 CSL_Ddr2Timing1::trrd
Specifies TRRD value: Minimum number of DDR2 EMIF cycles from Activate command to Activate command for a different bank, minus one

Uint8 CSL_Ddr2Timing1::twr
Specifies TWR value: Minimum number of DDR2 EMIF cycles from last write transfer to Pre-charge command, minus one

Uint8 CSL_Ddr2Timing1::twtr
Specifies the minimum number of DDR2 EMIF clock cycles from last DDR Write to DDR Read, minus one

3.3.8 CSL_Ddr2Timing2

Detailed Description
Timing2 structure to set the Timing2 register of DDR2 SDRAM.

Field Documentation

Uint8 CSL_Ddr2Timing2::tcke
Specifies the minimum number of DDR2 EMIF clock cycles between pado_mcke_o changes, minus one.

Uint8 CSL_Ddr2Timing2::todt
Specifies the minimum number of DDR2 EMIF clock cycles from ODT enable to write data driven for DDR2 SDRAM.

Uint8 CSL_Ddr2Timing2::trtp
Specifies the minimum number of DDR2 EMIF clock cycles from the last Read command to a Pre-charge command for DDR2 SDRAM, minus one.

Uint8 CSL_Ddr2Timing2::tsxnr
Specifies the minimum number of DDR2 EMIF clock cycles from Self-Refresh exit to any command other than a Read command, minus one.

Uint8 CSL_Ddr2Timing2::tsxrd
Specifies the minimum number of DDR2 EMIF clock cycles from Self-Refresh exit to a Read command for DDR SDRAM, minus one.

3.3.9 CSL_Ddr2Settings

Detailed Description
This structure contains the fields to set the DDR2 SDRAM. All fields needed for DDR2 SDRAM settings are present in this structure.

Field Documentation

CSL_Ddr2CasLatency CSL_Ddr2Settings::casLatncy
CAS Latency

CSL_Ddr2IntBank CSL_Ddr2Settings::ibank
Defines number of banks inside connected SDRAM devices

CSL_Ddr2PageSize CSL_Ddr2Settings::pageSize
Defines the internal page size of connected SDRAM devices

CSL_Ddr2Mode CSL_Ddr2Settings:: narrowMode
SDRAM data bus width

CSL_Ddr2Drive CSL_ Ddr2Settings:: ddrDrive
DDR SDRAM drive strength

3.3.10 CSL_Ddr2ModIdRev

Detailed Description
DDR2 Module ID and Revision structure is used for querying the DDR2 module Id and revision.

Field Documentation

Uint8 CSL_Ddr2ModIdRev::majRev
DDR2 EMIF Major Revision

Uint8 CSL_Ddr2ModIdRev::minRev
DDR2 EMIF Minor Revision

Uint16 CSL_Ddr2ModIdRev::modId
DDR2 EMIF Module ID

3.4 Enumerations

This section lists the enumerations available in the DDR2 module.

3.4.1 CSL_Ddr2CasLatency

enum CSL_Ddr2CasLatency
Enumeration for bit field CL of SDRAM Config Register.

Enumeration values:

CSL_DDR2_CAS_LATENCY_2	Cas Latency is 2
CSL_DDR2_CAS_LATENCY_3	Cas Latency is 3
CSL_DDR2_CAS_LATENCY_4	Cas Latency is 4
CSL_DDR2_CAS_LATENCY_5	Cas Latency is 5

3.4.2 CSL_Ddr2IntBank

enum CSL_Ddr2IntBank
Enumeration for bit field ibank of SDRAM Config Register.

Enumeration values:

CSL_DDR2_1_SDRAM_BANKS	DDR2 SDRAM has one internal bank
CSL_DDR2_2_SDRAM_BANKS	DDR2 SDRAM has two internal banks
CSL_DDR2_4_SDRAM_BANKS	DDR2 SDRAM has four internal bank
CSL_DDR2_8_SDRAM_BANKS	DDR2 SDRAM has eight internal banks

3.4.3 CSL_Ddr2PageSize

enum CSL_Ddr2PageSize
Enumeration for bit field pagesize of SDRAM Config Register.

Enumeration values:

CSL_DDR2_256WORD_8COL_ADDR	256-word pages requiring 8 column address bits
CSL_DDR2_512WORD_9COL_ADDR	512-word pages requiring 9 column address bits
CSL_DDR2_1024WORD_10COL_ADDR	1024-word pages requiring 10 column address bits
CSL_DDR2_2048WORD_11COL_ADDR	2048-word pages requiring 11 column address bits

3.4.4 CSL_Ddr2SelfRefresh

enum CSL_Ddr2SelfRefresh
Enumeration for bit field SR of SDRAM Config Register.

Enumeration values:

CSL_DDR2_SELF_REFRESH_DISABLE	Disables Self Refresh on DDR2
CSL_DDR2_SELF_REFRESH_ENABLE	Connected DDR2 SDRAM device will enter Self Refresh Mode and DDR2 EMIF enters Self Refresh State

3.4.5 CSL_Ddr2HwStatusQuery

enum CSL_Ddr2HwStatusQuery
Enumeration for queries passed to *CSL_ddr2GetHwStatus()*.
This is used to get the status of different operations

Enumeration values:

CSL_DDR2_QUERY_REV_ID	Get the DDR2 EMIF module ID and revision numbers. **Parameters:** (CSL_Ddr2ModIdRev *)
CSL_DDR2_QUERY_REFRESH_RATE	Get the EMIF refresh rate information **Parameters:** (Uint16 *)
CSL_DDR2_QUERY_SELF_REFRESH	Get self refresh bit value **Parameters:** (CSL_Ddr2SelfRefresh *)
CSL_DDR2_QUERY_IFRDY	Reflects the value on the IFRDY_ready port (active high) that defines whether the DDR PHY is ready for normal operation. **Parameters:** (Uint8 *)
CSL_DDR2_QUERY_ ENDIAN	Gets the the current endian of DDR2 emif from the SDRAM Status register. **Parameters:** (Uint8 *)

3.4.6 CSL_Ddr2HwControlCmd

enum CSL_Ddr2HwControlCmd
Enumeration for commands passed to *CSL_ddr2HwControl()*.
This is used to select the commands to control the operations existing setup of DDR2. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_DDR2_CMD_SELF_REFRESH	Self refresh enable or disable based on arg passed **Parameters:** (CSL_Ddr2SelfRefresh *)
CSL_DDR2_CMD_REFRESH_RATE	Enters the Refresh rate value **Parameters:** (Uint16 *)
CSL_DDR2_CMD_PRIO_RAISE	Number of memory transfers after which the DDR2 EMIF momentarily raises the priority of old commands in the VBUSM Command FIFO. **Parameters:** (Uint8 *)

3.4.7 CSL_Ddr2Mode

enum CSL_Ddr2Mode
Enumeration for bit field narrow_mode of SDRAM Config Register

Enumeration values:

CSL_DDR2_NORMAL_MODE	DDR2 SDRAM data bus width is 32 bits
CSL_DDR2_NARROW_MODE	DDR2 SDRAM data bus width is 16 bits

3.4.8 CSL_Ddr2Drive

enum CSL_Ddr2Drive
Enumeration for bit field ddr_drive of SDRAM Config Register

Enumeration values:

CSL_DDR2_NORM_DRIVE	DDR2 SDRAM data bus width is 32 bits
CSL_DDR2_WEAK_DRIVE	DDR2 SDRAM data bus width is 16 bits

3.5 Macros

**#define CSL_DDR2_CONFIG_DEFAULTS **
```
{ \
    CSL_DDR2_SDCFG_DEFAULT,        \
    CSL_DDR2_SDRFC_DEFAULT,        \
    CSL_DDR2_SDTIM1_DEFAULT,        \
    CSL_DDR2_SDTIM2_DEFAULT,        \
    CSL_DDR2_BPRIO_RESETVAL \
}
```
Default values for Config structure.

**#define CSL_DDR2_SETTING_DEFAULTS **
```
{  \
    (CSL_Ddr2CasLatency)CSL_DDR2_CAS_LATENCY_5, \
    (CSL_Ddr2IntBank)CSL_DDR2_4_SDRAM_BANKS, \
    (CSL_Ddr2PageSize)CSL_DDR2_256WORD_8COL_ADDR, \
    (CSL_Ddr2Mode)CSL_DDR2_NORMAL_MODE, \
    (CSL_Ddr2Drive)CSL_DDR2_NORM_DRIVE \
}
```
The default values of DDR2 SDRAM settings.

**#define CSL_DDR2_TIMING1_DEFAULTS **
```
{\
    (Uint8)CSL_DDR2_TIMING1_TRFC_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TRP_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TRCD_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TWR_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TRAS_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TRC_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TRRD_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING1_TWTR_DEFAULT \
}
```
The default values of DDR2 SDRAM Timing1 Control structure.

**#define CSL_DDR2_TIMING2_DEFAULTS **
```
{ \
    (Uint8)CSL_DDR2_TIMING2_T_ODT_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING2_TSXNR_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING2_TSXRD_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING2_TRTP_DEFAULT, \
    (Uint8)CSL_DDR2_TIMING2_TCKE_DEFAULT \
}
```
The default values of DDR2 SDRAM Timing2 Control structure.

#define CSL_DDR2_TIMING1_TRFC_DEFAULT 0x7F
#define CSL_DDR2_TIMING1_TRP_DEFAULT 0x07
#define CSL_DDR2_TIMING1_TRCD_DEFAULT 0x07
#define CSL_DDR2_TIMING1_TWR_DEFAULT 0x07
#define CSL_DDR2_TIMING1_TRAS_DEFAULT 0x1F
#define CSL_DDR2_TIMING1_TRC_DEFAULT 0x1F
#define CSL_DDR2_TIMING1_TRRD_DEFAULT 0x07
#define CSL_DDR2_TIMING1_TWTR_DEFAULT 0x03

The defaults of DDR2 SDRAM Timing1 Control structure

#define CSL_DDR2_TIMING2_T_ODT_DEFAULT 0x03
#define CSL_DDR2_TIMING2_TSXNR_DEFAULT 0x7F
#define CSL_DDR2_TIMING2_TSXRD_DEFAULT 0xFF
#define CSL_DDR2_TIMING2_TRTP_DEFAULT 0x07
#define CSL_DDR2_TIMING2_TCKE_DEFAULT 0x1F
The defaults of DDR2 SDRAM Timing2 Control structure

#define CSL_DDR2_SDCFG_DEFAULT (0x00008A20u)
#define CSL_DDR2_SDRFC_DEFAULT (0x00000753u)
#define CSL_DDR2_SDTIM1_DEFAULT (0xFFFFFFFBu)
#define CSL_DDR2_SDTIM2_DEFAULT (0x007FFFFFu)
The default values of SDRAM config, refresh, timing1 and timing2 registers,which are other than the reset values

3.6 Typedefs

typedef CSL_Ddr2Obj * CSL_Ddr2Handle
This is a pointer to CSL_Ddr2Obj and is passed as the first parameter to all DDR2 CSL APIs

Chapter 4
EDMA MODULE

Topics

4.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within EDMA module.

The EDMA controller handles all data transfers between the level-two (L2) cache/memory controller and the device peripherals.These data transfers include cache servicing, non-cacheable memory accesses, user-programmed data transfers, and host accesses. The EDMA supports up to 64-event channels and 4 QDMA channels. The EDMA consists of a scalable Parameter RAM (PaRAM) that supports flexible ping-pong, circular buffering, channel-chaining, auto-reloading, and memory protection. The EDMA allows movement of data to/from any addressable memory spaces, including internal memory (L2 SRAM), peripherals, and external memory.

4.2 Functions

This section lists the functions available in the EDMA module.

4.2.1 CSL_edma3Init

CSL_Status CSL_edma3Init (CSL_Edma3Context * *pContext*)

Description
This is the initialization function for the EDMA CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

`pContext`	`Pointer to module-context. As edma doesn't have any context based information user is expected to pass NULL.`

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
if (CSL_edma3Init(NULL) != CSL_SOK) {
    return;
}
...
```

4.2.2 CSL_edma3Open

CSL_Edma3Handle CSL_edma3Open (

CSL_Edma3Obj *	*pEdmaObj*,
CSL_InstNum	*edmaNum*,
CSL_Edma3ModuleAttr *	*pAttr*,
CSL_Status *	*pStatus*

)

Description
This function opens the EDMA3 CSL. It returns a handle to the edma instance. This handle is passed to all other CSL APIs, as the reference to the EDMA instance.

Arguments

`pEdmaObj`	`Pointer to EDMA Module Object`
`edmaNum`	`Instance of EDMA to be opened`
`pAttr`	`EDMA Attribute pointer`
`pStatus`	`Status of the function call`

Return Value
`CSL_Edma3Handle`

- Valid Edma handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The EDMA must be successfully initialized via *CSL_edma3Init()* before calling this function.

Post Condition
1. The status is returned in the pStatus variable. If status returned is

- `CSL_SOK` - Valid EDMA handle is returned
- `CSL_ESYS_FAIL` -The EDMA instance is invalid
- `CSL_ESYS_INVPARAMS` -The object passed is invalid

2. EDMA object structure is populated.

Modifies
1. The status variable
2. EDMA object structure

Example

```
CSL_Edma3Handle     hModule;
CSL_Edma3Obj        edmaObj;
CSL_Status          status;
...
// Module Initialization
status = CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);
...
```

4.2.3 CSL_edma3Close

CSL_Status CSL_edma3Close (CSL_Edma3Handle *hEdma*)

Description
This is a module level close required to invalidate the module handle. The module handle must not be used after this API call.

Arguments

```
hEdma          Handle to the EDMA Instance
```

Return Value
`CSL_Status`

- `CSL_SOK` - EDMA is closed successfully.
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
The functions *CSL_edma3Init()* and *CSL_edma3Open()* have to be called in order before calling this function.

Post Condition
The EDMA CSL APIs can not be called until the EDMA CSL is reopened again using *CSL_edma3Open()*.

Modifies
CSL_edma3Obj structure values

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj, CSL_EDMA3, NULL, &status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Open Channels, setup transfers etc
...
// Close Module
status = CSL_edma3Close(hModule);
```

4.2.4 CSL_edma3HwSetup

CSL_Status CSL_edma3HwSetup	**(**	**CSL_Edma3Handle**	***hMod*,**
		CSL_Edma3HwSetup *	***setup***
	)		

Description
This function initializes the device registers with the appropriate values provided through the HwSetup Data structure. After the Setup is completed, the device is ready for operation. For information passed through the HwSetup data structure, refer CSL_Edma3HwSetup. This does the setup for all dma/qdma channels viz. the parameter entry mapping, the trigger word setting (if QDMA channels) and the event queue mapping of the channel.

Arguments

```
hMod            Edma module Handle

setup           Pointer to the setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` – Successful completion of hardware setup
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* must be called successfully in that order before calling this API.

Post Condition
EDMA registers are configured according to the hardware setup parameters.

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;

CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwQdmaChannelSetup qdmahwSetup[CSL_EDMA3_NUM_QDMACH] =
                             CSL_EDMA3_QDMACHANNELSETUP_DEFAULT;
CSL_Status                  status;


// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = &qdmahwSetup[0];
status = CSL_edma3HwSetup(hModule,&hwSetup);
...
```

4.2.5 CSL_edma3GetHwSetup

CSL_Status CSL_edma3GetHwSetup **(** **CSL_Edma3Handle** ***hMod*****,**
CSL_Edma3HwSetup * ***setup***
)

Description
It gets the hwsetup parameters of the all edma/qdma channels.

Arguments

`hMod`	`Edma Handle`
`setup`	`Pointer to the setup structure`

Return Value
`CSL_Status`

- `CSL_SOK` - Getting module setup is successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is Invalid

Pre Condition
The functions *CSL_edma3Init()*, *CSL_edma3Open()* must be called successfully in order before calling *CSL_edma3GetHwSetup()*.

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
None

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup,gethwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                              CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup   getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Status                   status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);
// Get Module Setup
```

```
gethwSetup.dmaChaSetup  = &getdmahwSetup[0];
gethwSetup.qdmaChaSetup = NULL;
status = CSL_edma3GetHwSetup(hModule,&gethwSetup);
...
```

4.2.6 CSL_edma3HwControl

CSL_Status CSL_edma3HwControl	**(**	**CSL_Edma3Handle**	***hMod*,**
		CSL_Edma3HwControlCmd	***cmd*,**
		void *	***cmdArg***
	)		

Description
This function takes a command with an optional argument and implements it. This function is used to carry out the different operations performed by EDMA.

Arguments

```
hMod        Edma module Handle

cmd         The command to this API which indicates the
            action to be taken

cmdArg      Pointer argument specific to the command
```

Return Value
`CSL_Status`

- `CSL_SOK` - Command execution successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVCMD` - The command passed is invalid

Pre Condition
The functions *CSL_edma3Init()*, *CSL_edma3Open()* must be called successfully in order before calling this API.

Post Condition
Edma registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
EDMA registers determined by the command

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3QueryInfo           info;
CSL_Edma3CmdDrae             regionAccess;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
```

```
CSL_Status                    status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Query Module Info
CSL_edma3GetHwStatus(hModule,CSL_EDMA3_QUERY_INFO,&info);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);
...
```

4.2.7 CSL_edma3GetHwStatus

CSL_Status CSL_edma3GetHwStatus	**(**	CSL_Edma3Handle	***hMod***,
		CSL_Edma3HwStatusQuery	***myQuery***,
		void *	***response***
	)		

Description
This function gets the status of the different operations or the current setup of EDMA module.

Arguments

```
hMod            Edma module handle

myQuery         Query to be performed

response        Pointer to buffer to return the data requested by
                the query passed
```

Return Value
`CSL_Status`

- `CSL_SOK` - Getting the status of edma is successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVQUERY` - The query passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
The functions *CSL_edma3Init()*, *CSL_edma3Open()* must be called successfully in order before calling this API. Argument type that can be void* casted and passed with a particular command refer to *CSL_Edma3HwStatusQuery*.

Post Condition
None

Modifies
The input argument "response" is modified.

Example
```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3QueryInfo           info;
CSL_Status                   status;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                               CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Query Module Info
CSL_edma3GetHwStatus(hModule,CSL_EDMA3_QUERY_INFO, &info);
...
```

4.2.8 CSL_edma3ccGetModuleBaseAddr

CSL_Status CSL_edma3ccGetModuleBaseAddr (

CSL_InstNum	***edmaNum*,**
CSL_Edma3ModuleAttr *	***pParam*,**
CSL_Edma3ModuleBaseAddress *	***pBaseAddress***
)	

Description
This function is used for getting the base-address of the EDMA module. This function will be called inside the *CSL_edma3Open()/ CSL_edma3ChannelOpen()* function call.
Note: This function is open for re-implemention if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

edmaNum	Specifies the instance of the edma to be opened
pParam	Module specific parameters

```
pBaseAddress      Pointer to baseaddress structure containing
                  base address details
```

Return Value
`CSL_Status`

- `CSL_SOK` - Successfully retrieved base address
- `CSL_ESYS_FAIL` - The instance number is invalid.
- `CSL_ESYS_INVPARAMS` – Invalid Base address structure

Pre Condition
None

Post Condition
Base Address structure is populated.

Modifies
- The status variable
- Base address structure is modified.

Example
```
CSL_Status                   status;
CSL_Edma3ModuleBaseAddress   baseAddress;
...
status = CSL_edma3ccGetModuleBaseAddr(CSL_EDMA3, NULL,
                                       &baseAddress);
...
```

4.2.9 CSL_edma3ChannelOpen

CSL_Edma3ChannelHandle CSL_edma3ChannelOpen(CSL_Edma3ChannelObj * ***pEdmaObj*****,**
CSL_InstNum ***edmaNum*****,**
CSL_Edma3ChannelAttr * ***pChAttr*****,**
CSL_Status * ***pStatus***
)

Description
The API returns a handle for the specified EDMA Channel for use. The channel can be re-opened anytime after it has been normally closed if so required. The handle returned by this call is input as an essential argument for many of the APIs described for this module.

Arguments

```
pEdmaObj       pointer to the object that holds reference to the
               channel instance of the Specified EDMA

edmaNum        EDMA Instance

pChAttr        Instance of Channel requested and Region
```

```
pStatus           Status of the function call
```

Return Value
`CSL_Edma3ChannelHandle`

The requested channel instance of the EDMA if the call is successful, else a NULL is returned.

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* must be invoked successfully in order before calling this API.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid channel handle is returned
- `CSL_ESYS_FAIL` - The EDMA instance is invalid
- `CSL_ESYS_INVPARAMS` - The Parameters passed are invalid

2. EDMA channel object structure is populated.

Modifies
1. The status variable
2. EDMA channel object structure

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3ChannelObj          chObj;
CSL_Edma3CmdDrae             regionAccess;
CSL_Edma3ChannelHandle       hChannel;
CSL_Edma3ChannelAttr         chAttr;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                                CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status                   status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);
```

```
 // Channel 0 Open in context of Shadow region 0
 chAttr.regionNum = CSL_EDMA3_REGION_0;
 chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
 hChannel = CSL_edma3ChannelOpen(&chObj,
                                 CSL_EDMA3,
                                 &chAttr,
                                 &status);

 // Setup a Parameter Entry
  ...

 // Manually trigger the Channel

CSL_edma3HwChannelControl(hChannel,
                          CSL_EDMA3_CMD_CHANNEL_SET,NULL);

// Close Channel
CSL_edma3ChannelClose(hChannel);
...
```

4.2.10 CSL_edma3ChannelClose

CSL_Status CSL_edma3ChannelClose (CSL_Edma3ChannelHandle *hEdma*)

Description
This function marks the channel cannot be accessed any more using the handle. CSL for the EDMA channel need to be reopened before using any edma channel.

Arguments

hEdma	Handle to the requested channel

Return Value
CSL_Status

- CSL_SOK - Edma channel is closed successfully.
- CSL_ESYS_BADHANDLE - The handle passed is invalid

Pre Condition
The functions *CSL_edma3Init(), CSL_edma3Open(), CSL_edma3ChannelOpen()* must be invoked successfully in order before calling this API.

Post Condition
The edma channel related CSL APIs can not be called until the edma channel is reopened again using *CSL_edma3ChannelOpen()*.

Modifies
CSL_Edma3ChannelObj structure values.

Example

```
CSL_Edma3Handle          hModule;
CSL_Edma3HwSetup         hwSetup;
CSL_Edma3Obj             edmaObj;
```

```
CSL_Edma3ChannelObj      chObj;
CSL_Edma3CmdDrae         regionAccess;
CSL_Edma3ChannelHandle   hChannel;
CSL_Edma3ChannelAttr     chAttr;
CSL_Edma3HwDmaChannelSetup dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                         CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status               status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);


// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule, CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Setup a Parameter Entry
...

// Manually trigger the Channel
CSL_edma3HwChannelControl(hChannel,
                          CSL_EDMA3_CMD_CHANNEL_SET,NULL);

// Close Channel
status = CSL_edma3ChannelClose(hChannel);
...
```

4.2.11 CSL_edma3HwChannelSetupParam

CSL_Status CSL_edma3HwChannelSetupParam ([CSL_Edma3ChannelHandle](#) ***hEdma*****,**
Uint16 ***paramNum***
)

Description
This function sets up the channel to parameter entry mapping. This writes the DCHMAP[] / QCHMAP appropriately.

Arguments

hEdma	Channel Handle
paramNum	Parameter Entry Number

Return Value
CSL_Status

- CSL_SOK - Channel setup param successful
- CSL_ESYS_BADHANDLE - The handle passed is invalid
- CSL_ESYS_INVPARAMS - The parameters passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API.

Post Condition
Channel to parameter entry is configured.

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
Uint16                      paramNum;
CSL_Status                  status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
```

```
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Set the parameter entry number to channel
paramNum = 100;
status = CSL_edma3HwChannelSetupParam(hChannel,paramNum);
...
```

4.2.12 CSL_edma3HwChannelSetupTriggerWord

CSL_Status CSL_edma3HwChannelSetupTriggerWord(CSL_Edma3ChannelHandle *hEdma*, Uint8 *triggerWord*)

Description
Programs the QDMA channel triggerword. This writes the QCHMAP appropriately.

Arguments

```
hEdma           Channel Handle

triggerWord     Trigger word
```

Return Value
`CSL_Status`

- `CSL_SOK` - Channel setup triggerword is successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling *CSL_edma3HwChannelSetupTriggerWord()*.

Post Condition
Sets up the QDMA Channel to trigger Word

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
```

```
CSL_Status                    status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);


// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_QCHA_0;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Sets up the QDMA Channel 0 trigger Word to 3rd trigger word
status = CSL_edma3HwChannelSetupTriggerWord(hChannel,3);
...
```

4.2.13 CSL_edma3HwChannelSetupQue

CSL_Status CSL_edma3HwChannelSetupQue (CSL_Edma3ChannelHandle *hEdma*, CSL_Edma3Que *evtQue*)

Description
This function programs the channel to Queue mapping. This writes the DMAQNUM/QDAMQNUM appropriately.

Arguments

```
hEdma          Channel Handle

evtQue         Event Queue name
```

Return Value
`CSL_Status`

- `CSL_SOK` - Channel setup queue successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API.

Post Condition
Sets up the channel to Queue mapping

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                            CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status                  status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Set up the channel to que mapping
status = CSL_edma3HwChannelSetupQue(hChannel,CSL_EDMA3_QUE_3);
...
```

4.2.14 CSL_edma3GetHwChannelSetupParam

**CSL_Status CSL_edma3GetHwChannelSetupParam(CSL_Edma3ChannelHandle *hEdma*,
Uint16 * *paramNum*
)**

Description
This function obtains the Channel to Parameter Set mapping. This reads the DCHMAP/QCHMAP appropriately.

Arguments

```
hEdma            Channel Handle
```

```
    paramNum        Pointer to parameter entry
```

Return Value
`CSL_Status`

- `CSL_SOK` - Retrieving the parameter entry number to which a channel is mapped
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
The functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling *CSL_edma3GetHwChannelSetupParam()*.

Post Condition
None

Modifies
None

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
Uint16                      paramNum;
CSL_Status                  status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Get the parameter entry number to which a channel is mapped
status = CSL_edma3GetHwChannelSetupParam(hChannel,&paramNum);
...
```

4.2.15 CSL_edma3GetHwChannelSetupTriggerWord

CSL_Status CSL_edma3GetHwChannelSetupTriggerWord(**CSL_Edma3ChannelHandle** ***hEdma*****,**
Uint8 * ***triggerWord***
)

Description
This function read the QDMA channel triggerword. This reads the QCHMAP to obtain the trigger word appropriately.

Arguments

```
hEdma           Channel Handle

triggerWord     Pointer to Trigger word
```

Return Value
`CSL_Status`

- `CSL_SOK` - Retrieving the parameter entry number to which a channel is mapped
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling *CSL_edma3GetHwChannelSetupTriggerWord()*.

Post Condition
None

Modifies
None

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3ChannelObj          chObj;
CSL_Edma3ChannelHandle       hChannel;
CSL_Edma3ChannelAttr         chAttr;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
Uint8                        triggerWord;
CSL_Status                   status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
```

```
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_QCHA_0;
hChannel = CSL_edma3ChannelOpen(&chObj,
                        CSL_EDMA3,
                        &chAttr,
                        &status);

// Get the trigger word programmed for a channel
CSL_edma3GetHwChannelSetupTriggerWord(hChannel,&triggerWord);
...
```

4.2.16 CSL_edma3GetHwChannelSetupQue

CSL_Status CSL_edma3GetHwChannelSetupQue (CSL_Edma3ChannelHandle *hEdma*, CSL_Edma3Que * *evtQue*)

Description
This function obtains the channel to queue map for the channel. This reads the DMAQNUM / QDAMQNUM appropriately.

Arguments

hEdma	Channel Handle
evtQue	Pointer to Event Queue structure

Return Value
CSL_Status

- CSL_SOK - Retrieving the queue to which a channel is mapped
- CSL_ESYS_BADHANDLE - The handle passed is invalid
- CSL_ESYS_INVPARAMS –The parameter is Invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling *CSL_edma3GetHwChannelSetupQue()*.

Post Condition
None

Modifies
None

Example

```
CSL_Edma3Handle            hModule;
CSL_Edma3HwSetup           hwSetup;
```

```
CSL_Edma3Obj                  edmaObj;
CSL_Edma3ChannelObj           chObj;
CSL_Edma3ChannelHandle        hChannel;
CSL_Edma3ChannelAttr          chAttr;
CSL_Edma3HwDmaChannelSetup    dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                               CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3Que                  evtQue;
CSL_Status                    status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Get the que to which a channel is mapped
CSL_edma3GetHwChannelSetupQue(hChannel,&evtQue);
...
```

4.2.17 CSL_edma3HwChannelControl

CSL_Status CSL_edma3HwChannelControl(CSL_Edma3ChannelHandle ***hChannel*,**
CSL_Edma3HwChannelControlCmd ***cmd*,**
void * ***cmdArg***
)

Description
This function takes a command with an optional argument and implements it. This function is used to carry out the different operations performed by EDMA.

Arguments

```
hChannel         Channel Handle

cmd              The command to this API which indicates the
                 action to be taken

cmdArg           Pointer argument specific to the command
```

Return Value
CSL_Status

- CSL_SOK - Command execution successful
- CSL_ESYS_BADHANDLE - The handle passed is invalid
- CSL_ESYS_INVCMD - The command passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API. If a Shadow region is used, then care must be taken to set the DRAE.

Post Condition
EDMA registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
EDMA registers determined by the command

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3CmdIntr            regionIntr;
CSL_Edma3CmdDrae            regionAccess;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status                  status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);

// Interrupt Enable (Bits 0-11)  for the Shadow Region 0.
regionIntr.region =  CSL_EDMA3_REGION_0  ;
regionIntr.intr  =   0x0FFF ;
```

```
    regionIntr.intrh  =  0x0000 ;
    CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_INTR_ENABLE,
                       &regionIntr);

    // Channel 0 Open in context of Shadow region 0
    chAttr.regionNum = CSL_EDMA3_REGION_0;
    chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
    hChannel = CSL_edma3ChannelOpen(&chObj,
                            CSL_EDMA3,
                            &chAttr,
                            &status);

    // Enable Channel(if the channel is meant for external event)
    // This step is not required if the channel is chained to or
       manually triggered.

   CSL_edma3HwChannelControl(hChannel,CSL_EDMA3_CMD_CHANNEL_ENABLE,\
                             NULL);
   ...
```

4.2.18 CSL_edma3GetHwChannelStatus

CSL_Status CSL_edma3GetHwChannelStatus(CSL_Edma3ChannelHandle ***hEdma***,
CSL_Edma3HwChannelStatusQuery ***myQuery***,
void * ***response***
)

Description
This function gets the status of the different operations or the current setup of EDMA module.

Arguments

`hEdma`	`Channel Handle`
`myQuery`	`Query to be performed`
`response`	`Pointer to buffer to return the data requested by the query passed`

Return Value
`CSL_Status`

- `CSL_SOK` - Getting the EDMA channel status is successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVQUERY` - The query passed is invalid
- `CSL_ESYS_INVPARAMS` – The parameter passed is invalid

Pre Condition
The functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API. If a Shadow region is used, then care must be taken to set the DRAE.

Post Condition
None

Modifies
The input argument "response" is modified.

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status                  status;
Bool                        errStat;


// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Enable Channel(  .. )
...
CSL_edma3HwChannelControl(hChannel,
                          CSL_EDMA3_CMD_CHANNEL_ENABLE, NULL);

// Obtain Channel Error Status
CSL_edma3GetHwChannelStatus(hChannel,
                            CSL_EDMA3_QUERY_CHANNEL_ERR,
                            &errStat);
...
```

4.2.19 CSL_edma3GetParamHandle

CSL_Edma3ParamHandle CSL_edma3GetParamHandle(CSL_Edma3ChannelHandle *hEdma*,
Int16 *paramNum*,
CSL_Status * *status*
)

Description
This function acquires the PaRAM entry as specified by the argument.

Arguments

`hEdma`	`Channel Handle`
`paramNum`	`Parameter RAM (PaRAM) entry number`
`status`	`Status of the function call`

Return Value
`CSL_Edma3ParamHandle`

- Valid PaRAM handle will be returned if status value is equal to CSL_SOK.

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid channel handle is returned
- `CSL_ESYS_INVPARAMS` - The param set number is invalid
- `CSL_ESYS_BADHANDLE` – The handle passed is invalid.

Modifies
None

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3ChannelObj          chObj;
CSL_Edma3CmdIntr             regionIntr;
CSL_Edma3CmdDrae             regionAccess;
CSL_Edma3ChannelHandle       hChannel;
CSL_Edma3ChannelAttr         chAttr;
CSL_Edma3ParamHandle         hParamBasic;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                              CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
```

```
CSL_Status                    status;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);

 // Interrupt Enable (Bits 0-11)  for the Shadow Region 0.
 regionIntr.region =  CSL_EDMA3_REGION_0  ;
 regionIntr.intr  =   0x0FFF ;
 regionIntr.intrh  =  0x0000 ;
 CSL_edma3HwControl(hModule,
                   CSL_EDMA3_CMD_INTR_ENABLE,&regionIntr);

 // Channel 0 Open in context of Shadow region 0
 chAttr.regionNum = CSL_EDMA3_REGION_0;
 chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
 hChannel = CSL_edma3ChannelOpen(&chObj,
                        CSL_EDMA3,
                        &chAttr,
                        &status);

 // Obtain a handle to parameter entry 0
 hParamBasic = CSL_edma3GetParamHandle(hChannel,0,NULL);
 ...
```

4.2.20 CSL_edma3ParamSetup

CSL_Status CSL_edma3ParamSetup **(** **CSL_Edma3ParamHandle** ***hParamHndl*****,**
CSL_Edma3ParamSetup ***** ***setup***
)

Description
This function configures the EDMA Parameter RAM (PaRAM) entry using the values passed in through the PaRAM setup structure.

Arguments

```
hParamHndl       Handle to the PaRAM entry

setup            Pointer to PaRAM setup structure
```

Return Value
CSL_Status

- CSL_SOK - PaRAM setup successful
- CSL_ESYS_BADHANDLE - The handle passed is invalid
- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
Functions *CSL_edma3Init()*, *CSL_edma3Open()* and *CSL_edma3ChannelOpen()* must be called successfully in order before calling this API.

Post Condition
Configures the EDMA PaRAM entry

Modifies
Parameter entry

Example

```
CSL_Edma3Handle            hModule;
CSL_Edma3HwSetup           hwSetup;
CSL_Edma3Obj               edmaObj;
CSL_Edma3ChannelObj        chObj;
CSL_Edma3CmdIntr           regionIntr;
CSL_Edma3CmdDrae           regionAccess;
CSL_Edma3ChannelHandle     hChannel;
CSL_Edma3ParamHandle       hParamBasic;
CSL_Edma3ParamSetup        myParamSetup;
CSL_Edma3ChannelAttr       chAttr;
CSL_Edma3HwDmaChannelSetup dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                            CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Status                 status;
Uint8                      srcBuff1[512];
Uint8                      dstBuff1[512];

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);

// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = NULL;
CSL_edma3HwSetup(hModule,&hwSetup);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.drae =   0xFFFF ;
regionAccess.draeh =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_DMAREGION_ENABLE, \
                   &regionAccess);
```

```
 // Interrupt Enable (Bits 0-11)  for the Shadow Region 0.
 regionIntr.region =  CSL_EDMA3_REGION_0  ;
 regionIntr.intr  =   0x0FFF ;
 regionIntr.intrh  =  0x0000 ;
 CSL_edma3HwControl(hModule,
                   CSL_EDMA3_CMD_INTR_ENABLE,&regionIntr);

 // Channel 0 Open in context of Shadow region 0
 chAttr.regionNum = CSL_EDMA3_REGION_0;
 chAttr.chaNum = CSL_EDMA3_CHA_DSP_EVT;
 hChannel = CSL_edma3ChannelOpen(&chObj,
                        CSL_EDMA3,
                        &chAttr,
                        &status);

 // Obtain a handle to parameter entry 0
hParamBasic = CSL_edma3GetParamHandle(hChannel,0,NULL);

// Setup the first param Entry (Ping buffer)
myParamSetup.option = CSL_EDMA3_OPT_MAKE(CSL_EDMA3_ITCCH_DIS, \
                                   CSL_EDMA3_TCCH_DIS, \
                                   CSL_EDMA3_ITCINT_DIS, \
                                   CSL_EDMA3_TCINT_EN,\
                                   0,CSL_EDMA3_TCC_NORMAL,\
                                   CSL_EDMA3_FIFOWIDTH_NONE, \
                                   CSL_EDMA3_STATIC_DIS, \
                                   CSL_EDMA3_SYNC_A, \
                                   CSL_EDMA3_ADDRMODE_INCR, \
                                   CSL_EDMA3_ADDRMODE_INCR);
myParamSetup.srcAddr = (Uint32)srcBuff1;
myParamSetup.aCntbCnt = CSL_EDMA3_CNT_MAKE(256,1);
myParamSetup.dstAddr = (Uint32)dstBuff1;
myParamSetup.srcDstBidx = CSL_EDMA3_BIDX_MAKE(1,1);
myParamSetup.linkBcntrld = CSL_EDMA3_LINKBCNTRLD_MAKE
                                   (CSL_EDMA3_LINK_NULL,0);
myParamSetup.srcDstCidx = CSL_EDMA3_CIDX_MAKE(0,1);
myParamSetup.cCnt = 1;
CSL_edma3ParamSetup(hParamBasic,&myParamSetup);
...
```

4.2.21 CSL_edma3ParamWriteWord

CSL_Status CSL_edma3ParamWriteWord (**CSL_Edma3ParamHandle** ***hParamHndl*****,**
Uint16 ***wordOffset*****,**
Uint32 ***word***
)

Description
This is for the ease of QDMA channels. Once the QDMA channel transfer is triggered, subsequent triggers may be done with only writing the modified words in the parameter RAM (PaRAM) entry along with the trigger word. This API is expected to achieve this purpose. Most usage scenarios, the user should not be writing more than the trigger word entry.

Arguments

`hParamHndl`	Handle to the PaRAM entry
`wordOffset`	Word offset in the 8 word parameter entry
`word`	Word to be written

Return Value
`CSL_Status`

- `CSL_SOK` - PaRAM Write Word successful.
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Functions *CSL_edma3Init(), CSL_edma3Open()* and *CSL_edma3ChannelOpen(), CSL_edma3GetParamHandle(), CSL_edma3ParamSetup()* must be called successfully in order before calling this API. The main setup structure consists of pointers to sub-structures. The user has to allocate space for and fill in the parameter (PaRAM) setup structure.

Post Condition
Configure trigger word

Modifies
None

Example

```
CSL_Edma3Handle              hModule;
CSL_Edma3HwSetup             hwSetup;
CSL_Edma3Obj                 edmaObj;
CSL_Edma3ParamHandle         hParamBasic;
CSL_Edma3ChannelObj          chObj;
CSL_Edma3CmdIntr             regionIntr;
CSL_Edma3CmdQrae             regionAccess;
CSL_Edma3ChannelHandle       hChannel;
CSL_Edma3ParamSetup          myParamSetup;
CSL_Edma3ChannelAttr         chAttr;
CSL_Status                   status;
CSL_Edma3HwDmaChannelSetup   dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwQdmaChannelSetup qdmahwSetup[CSL_EDMA3_NUM_QDMACH] =
                             CSL_EDMA3_QDMACHANNELSETUP_DEFAULT;

// Module Initialization
CSL_edma3Init(NULL);

// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);


// Module Setup
hwSetup.dmaChaSetup  = &dmahwSetup[0];
hwSetup.qdmaChaSetup = &qdmahwSetup[0];;
```

```
CSL_edma3HwSetup(hModule,&hwSetup);

// DRAE Enable(Bits 0-15) for the Shadow Region 0.
regionAccess.region = CSL_EDMA3_REGION_0 ;
regionAccess.qrae =   0x000F ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_QDMAREGION_ENABLE, \
                   &regionAccess);

// Interrupt Enable (Bits 0-11) for the Shadow Region 0.
regionIntr.region =  CSL_EDMA3_REGION_0  ;
regionIntr.intr  =   0x0FFF ;
regionIntr.intrh  =  0x0000 ;
CSL_edma3HwControl(hModule,CSL_EDMA3_CMD_INTR_ENABLE,
                &regionIntr);

// Channel 0 Open in context of Shadow region 0
chAttr.regionNum = CSL_EDMA3_REGION_0;
chAttr.chaNum = CSL_EDMA3_QCHA_0;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

// Obtain a handle to parameter entry 0
hParamBasic = CSL_edma3GetParamHandle(hChannel,0,NULL);


// Setup the first param Entry (Ping buffer)
myParamSetup.option = CSL_EDMA3_OPT_MAKE(CSL_EDMA3_ITCCH_DIS, \
                                    CSL_EDMA3_TCCH_DIS, \
                                    CSL_EDMA3_ITCINT_DIS, \
                                    CSL_EDMA3_TCINT_EN,\
                                    0,CSL_EDMA3_TCC_NORMAL,\
                                    CSL_EDMA3_FIFOWIDTH_NONE, \
                                    CSL_EDMA3_STATIC_EN, \
                                    CSL_EDMA3_SYNC_A, \
                                    CSL_EDMA3_ADDRMODE_INCR, \
                                    CSL_EDMA3_ADDRMODE_INCR);
myParamSetup.srcAddr = (Uint32)srcBuff1;
myParamSetup.aCntbCnt = CSL_EDMA3_CNT_MAKE(256,1);
myParamSetup.dstAddr = (Uint32)dstBuff1;
myParamSetup.srcDstBidx = CSL_EDMA3_BIDX_MAKE(1,1);
myParamSetup.linkBcntrld = CSL_EDMA3_LINKBCNTRLD_MAKE
                                      (CSL_EDMA3_LINK_NULL,0);
myParamSetup.srcDstCidx = CSL_EDMA3_CIDX_MAKE(0,1);
myParamSetup.cCnt = 1;
CSL_edma3ParamSetup(hParamBasic,&myParamSetup);

// Enable Channel
CSL_edma3HwChannelControl(hChannel,
                          CSL_EDMA3_CMD_CHANNEL_ENABLE, NULL);
// Write trigger word
CSL_edma3ParamWriteWord(hParamBasic,7,myParamSetup.cCnt);
...
```

4.3 Data Structures

This section lists the data structures available in the EDMA module.

4.3.1 CSL_Edma3Obj

Detailed Description

This object contains the reference to the instance of EDMA Module opened using the *CSL_edma3Open()*. A pointer to this object is passed to all EDMA Module level CSL APIs.

Field Documentation

CSL_InstNum CSL_Edma3Obj::instNum

This is the instance of module number i.e. CSL_EDMA3

CSL_Edma3ccRegsOvly CSL_Edma3Obj::regs

This is a pointer to the EDMA Channel Controller registers of the module requested.

4.3.2 CSL_Edma3ParamSetup

Detailed Description

Edma Parameter RAM (PaRAM) setup Structure. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3ParamSetup()*. This structure is used to program the PaRAM Set for EDMA/QDMA. The macros can be used to assign values to the fields of the structure. The setup structure should be setup using the macros provided OR as per the bit descriptions in the user guide.

Field Documentation

Uint32 CSL_Edma3ParamSetup::aCntbCnt

Lower 16 bits are A count and upper 16 bits are B count

Uint32 CSL_Edma3ParamSetup::cCnt

C count

Uint32 CSL_Edma3ParamSetup::dstAddr

Specifies the destination address

Uint32 CSL_Edma3ParamSetup::linkBcntrld

Lower 16 bits are link of the next PaRAM entry and upper 16 bits are B count reload

Uint32 CSL_Edma3ParamSetup::option

Channel Options

Uint32 CSL_Edma3ParamSetup::srcAddr

Specifies the source address

Uint32 CSL_Edma3ParamSetup::srcDstBidx

Lower 16 bits are source B index and upper 16 bits are destination B index

Uint32 CSL_Edma3ParamSetup::srcDstCidx

Lower 16 bits are source C index and upper 16 bits are destination C index

4.3.3 CSL_Edma3ChannelObj

Detailed Description
Edma channel object structure. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3ChannelOpen().* The *CSL_edma3ChannelOpen()* updates all the members of the data structure and returns the objects address as a *CSL_Edma3ChannelHandle*. The *CSL_Edma3ChannelHandle* is used in all subsequent function calls.

Field Documentation

Int CSL_Edma3ChannelObj::chaNum
Channel Number being requested

Int CSL_Edma3ChannelObj::edmaNum
EDMA instance whose channel is being requested

Int CSL_Edma3ChannelObj::region
Region number to which the channel belongs

CSL_Edma3ccRegsOvly CSL_Edma3ChannelObj::regs
Pointer to the EDMA Channel Controller module register overlay structure

4.3.4 CSL_Edma3CtrlErrStat

Detailed Description
EDMA controller error status. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetControllerError() /CSL_edma3GetHwStatus().*

Field Documentation

CSL_BitMask16 CSL_Edma3CtrlErrStat::error
Bit Mask of the Queue Threshold Errors

Bool CSL_Edma3CtrlErrStat::exceedTcc
Status to know whether number of permissible outstanding TCCs is exceeded

4.3.5 CSL_Edma3QueryInfo

Detailed Description
EDMA controller information. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetInfo() /CSL_edma3GetHwStatus().*

Field Documentation

Uint32 CSL_Edma3QueryInfo::config
Channel Controller Configuration obtained from the CCCFG register

Uint32 CSL_Edma3QueryInfo::revision
Revision/Peripheral id of the EDMA3 Channel Controller

4.3.6 CSL_Edma3ActivityStat

Detailed Description
EDMA channel controller activity status. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetActivityStatus() / CSL_edma3GetHwStatus().*

Field Documentation

Bool CSL_Edma3ActivityStat::active
Indicates if the Channel Controller is active at all

Bool CSL_Edma3ActivityStat::evtActive
Indicates whether any EDMA events are active

Uint16 CSL_Edma3ActivityStat::outstandingTcc
Number of outstanding completion requests

Bool CSL_Edma3ActivityStat::qevtActive
Indicates whether any QDMA events are active

CSL_BitMask16 CSL_Edma3ActivityStat::queActive
BitMask of the queue active in the Channel Controller

Bool CSL_Edma3ActivityStat::trActive
Indicates whether the TR processing/submission logic is active

4.3.7 CSL_Edma3QueStat

Detailed Description
EDMA controller queue status. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetQueStatus() /CSL_edma3GetHwStatus()*.

Field Documentation

Bool CSL_Edma3QueStat::exceed
Output field: The number of valid entries in a queue has exceeded the threshold specified in QWMTHRA has been exceeded
.

Uint8 CSL_Edma3QueStat::numVal
Output field: Number of valid entries in Queue N

CSL_Edma3Que CSL_Edma3QueStat::que
Input field: Event Queue. This needs to be specified by the user before invocation of the above API

Uint8 CSL_Edma3QueStat::startPtr
Output field: Start pointer/Head of the queue

Uint8 CSL_Edma3QueStat::waterMark
Output field: The most entries that have been in Queue since reset/last time the watermark was cleared

4.3.8 CSL_Edma3CmdRegion

Detailed Description

EDMA control/query command structure for querying region specific attributes. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetHwStatus/CSL_edma3HwControl* with the relevant command.

Field Documentation

Int CSL_Edma3CmdRegion::region

Input field:- this field needs to be initialized by the user before issuing the query/command

CSL_BitMask32 CSL_Edma3CmdRegion::regionVal

Input/Output field. This needs to be filled by the user in case of issuing a COMMAND or it will be filled in by the CSL when used with a QUERY

4.3.9 CSL_Edma3CmdQrae

Detailed Description

EDMA control/query command structure for querying QDMA region access enable attributes. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetHwStatus/CSL_edma3HwControl* with the relevant command.

Field Documentation

CSL_BitMask32 CSL_Edma3CmdQrae::qrae

This needs to be filled by the user in case of issuing a COMMAND or it will be filled in by the CSL when used with a QUERY

Int CSL_Edma3CmdQrae::region

This field needs to be initialized by the user before issuing the query/command

4.3.10 CSL_Edma3CmdIntr

Detailed Description

EDMA control/query control command structure for issuing commands for interrupt related APIs An object of this type is allocated by the user and its address is passed to the Control API.

Field Documentation

CSL_BitMask32 CSL_Edma3CmdIntr::intr

Input/Output field: - this needs to be filled by the user in case of issuing a COMMAND or it will be filled in by the CSL when used with a QUERY

CSL_BitMask32 CSL_Edma3CmdIntr::intrh

Input/Output: - this needs to be filled by the user in case of issuing a COMMAND or it will be filled in by the CSL when used with a QUERY

Int CSL_Edma3CmdIntr::region

Input field: - this field needs to be initialized by the user before issuing the query/command

4.3.11 CSL_Edma3CmdDrae

Detailed Description

EDMA command structure for setting region specific attributes. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetHwStatus*.

Field Documentation

CSL_BitMask32 CSL_Edma3CmdDrae::drae
DRAE Setting for the region

CSL_BitMask32 CSL_Edma3CmdDrae::draeh
DRAEH Setting for the region

Int CSL_Edma3CmdDrae::region
This field needs to be initialized by the user before issuing the command specifying the region for which attributes need to be set

4.3.12 CSL_Edma3CmdQuePri

Detailed Description
EDMA command structure used for setting event queue priority level.
An object of this type is allocated by the user and its address is passed as a parameter to the CSL_edma3HwControl API.

Field Documentation

CSL_Edma3QuePri CSL_Edma3CmdQuePri::pri
Queue priority

CSL_Edma3Que CSL_Edma3CmdQuePri::que
Specifies the queue that needs a priority change

4.3.13 CSL_Edma3CmdQueThr

Detailed Description
EDMA command structure used for setting event queue threshold level. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3HwControl* API.

Field Documentation

CSL_Edma3Que CSL_Edma3CmdQueThr::que
Specifies the Queue that needs a change in the threshold setting

CSL_Edma3QueThr CSL_Edma3CmdQueThr::threshold
Queue threshold setting

4.3.14 CSL_Edma3ModuleBaseAddress

Detailed Description
This will have the base-address information for the module instance.

Field Documentation

CSL_Edma3ccRegsOvly CSL_Edma3ModuleBaseAddress::regs
Base-address of the peripheral registers

4.3.15 CSL_Edma3ChannelAttr

Detailed Description
EDMA channel parameter structure. This is used for opening a channel.

Field Documentation

Int CSL_Edma3ChannelAttr::chaNum
Channel number

Int CSL_Edma3ChannelAttr::regionNum
Region Number

4.3.16 CSL_Edma3ChannelErr

Detailed Description
Edma channel error structure. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetChannelError() /CSL_edma3GetHwStatus()/ CSL_edma3ChannelErrorClear() /CSL_edma3HwChannelControl()*.

Field Documentation

Bool CSL_Edma3ChannelErr::missed
A TRUE indicates an event is missed on this channel.

Bool CSL_Edma3ChannelErr::secEvt
A TRUE indicates an event that no events on this channel will be prioritized until this is cleared. This being TRUE does NOT necessarily mean it is an error. ONLY if both missed and ser are set, this kind of error needs to be cleared.

4.3.17 CSL_Edma3HwQdmaChannelSetup

Detailed Description
QDMA channel setup. An array of such objects are allocated by the user and address initialized in the CSL_Edma3HwSetup structure which is passed *CSL_edma3HwSetup().*

Field Documentation

Uint16 CSL_Edma3HwQdmaChannelSetup::paramNum
PaRAM set mapping for the channel.

CSL_Edma3Que CSL_Edma3HwQdmaChannelSetup::que
Queue number for the QDMA channel

Uint8 CSL_Edma3HwQdmaChannelSetup::triggerWord
Trigger word for the QDMA channels.

4.3.18 CSL_Edma3HwDmaChannelSetup

Detailed Description
EDMA channel setup. An array of such objects are allocated by the user and address initialized in the *CSL_Edma3HwSetup* structure which is passed *CSL_edma3HwSetup()*.

Field Documentation

CSL_Edma3Que CSL_Edma3HwDmaChannelSetup::que

Queue number for the channel

Uint16 CSL_Edma3HwDmaChannelSetup::paramNum
PaRAM set mapping for the channel

4.3.19 CSL_Edma3HwSetup

Detailed Description
This structure is used to setup or obtain existing setup of EDMA using *CSL_edma3HwSetup ()* and *CSL_edma3GetHwSetup ()* respectively.

Field Documentation

CSL_Edma3HwDmaChannelSetup* CSL_Edma3HwSetup::dmaChaSetup
Pointer to Edma Hw Channel setup structure.

CSL_Edma3HwQdmaChannelSetup* CSL_Edma3HwSetup::qdmaChaSetup
Pointer to QDMA channel setup structure

4.3.20 CSL_Edma3MemFaultStat

Detailed Description
Edma memory protection fault error status. An object of this type is allocated by the user and its address is passed as a parameter to the *CSL_edma3GetMemoryFaultError() / CSL_edma3GetHwStatus()* with the relevant command

Field Documentation
Uint32 CSL_Edma3MemFaultStat :: addr
Memory Protection Fault Address

CSL_BitMask16 CSL_Edma3MemFaultStat :: error
Bit Mask of the Errors

Uint16 CSL_Edma3MemFaultStat :: fid
Faulted ID

4.4 Enumerations

4.4.1 CSL_Edma3QuePri

enum CSL_Edma3QuePri
Enumeration for System priorities.
This is used for Setting up the Queue Priority level.

Enumeration values:

CSL_EDMA3_QUE_PRI_0	System priority level 0
CSL_EDMA3_QUE_PRI_1	System priority level 1
CSL_EDMA3_QUE_PRI_2	System priority level 2
CSL_EDMA3_QUE_PRI_3	System priority level 3
CSL_EDMA3_QUE_PRI_4	System priority level 4
CSL_EDMA3_QUE_PRI_5	System priority level 5
CSL_EDMA3_QUE_PRI_6	System priority level 6
CSL_EDMA3_QUE_PRI_7	System priority level 7

4.4.2 CSL_Edma3QueThr

enum CSL_Edma3QueThr
Enumeration for EDMA Queue Thresholds.
This is used for Setting up the Queue thresholds.

Enumeration values:

CSL_EDMA3_QUE_THR_0	EDMA Queue Threshold 0
CSL_EDMA3_QUE_THR_1	EDMA Queue Threshold 1
CSL_EDMA3_QUE_THR_2	EDMA Queue Threshold 2
CSL_EDMA3_QUE_THR_3	EDMA Queue Threshold 3
CSL_EDMA3_QUE_THR_4	EDMA Queue Threshold 4
CSL_EDMA3_QUE_THR_5	EDMA Queue Threshold 5
CSL_EDMA3_QUE_THR_6	EDMA Queue Threshold 6
CSL_EDMA3_QUE_THR_7	EDMA Queue Threshold 7
CSL_EDMA3_QUE_THR_8	EDMA Queue Threshold 8
CSL_EDMA3_QUE_THR_9	EDMA Queue Threshold 9
CSL_EDMA3_QUE_THR_10	EDMA Queue Threshold 10
CSL_EDMA3_QUE_THR_11	EDMA Queue Threshold 11
CSL_EDMA3_QUE_THR_12	EDMA Queue Threshold 12
CSL_EDMA3_QUE_THR_13	EDMA Queue Threshold 13
CSL_EDMA3_QUE_THR_14	EDMA Queue Threshold 14
CSL_EDMA3_QUE_THR_15	EDMA Queue Threshold 15
CSL_EDMA3_QUE_THR_16	EDMA Queue Threshold 16
CSL_EDMA3_QUE_THR_DISABLE	EDMA Queue Threshold Disable Errors

4.4.3 CSL_Edma3HwControlCmd

enum CSL_Edma3HwControlCmd
MODULE Level Commands

Enumeration values:

CSL_EDMA3_CMD_DMAREGION_ENABLE	Enables bits as specified in the argument passed in DRAE/DRAEH. Please note: If bits are already set in DRAE/DRAEH this Control command will cause additional bits (as specified by the bitmask) to be set and does **Parameters:** *(CSL_Edma3CmdDrae*)*
CSL_EDMA3_CMD_DMAREGION_DISABLE	Disables bits as specified in the argument passed in DRAE/DRAEH **Parameters:** *(CSL_Edma3CmdDrae*)*
CSL_EDMA3_CMD_QDMAREGION_ENABLE	Enables bits as specified in the argument passed in QRAE. Please note:If bits are already set in QRAE/QRAEH this Control command will cause additional bits (as specified by the bitmask) to be set and does. **Parameters:** *(CSL_Edma3CmdQrae*)*
CSL_EDMA3_CMD_QDMAREGION_DISABLE	Disables bits as specified in the argument passed in QRAE **Parameters:** *(CSL_Edma3CmdQrae*)*
CSL_EDMA3_CMD_QUEPRIORITY_SET	Programming QUEPRI register with the specified priority **Parameters:** *(CSL_Edma3CmdQuePri*)*
CSL_EDMA3_CMD_QUETHRESHOLD_SET	Programming QUEUE Threshold levels **Parameters:** *(CSL_Edma3CmdQueThr*)*
CSL_EDMA3_CMD_ERROR_EVAL	Sets the EVAL bit in the EEVAL register **Parameters:** *(None)*
CSL_EDMA3_CMD_INTRPEND_CLEAR	Clears specified (Bitmask) pending interrupt at Module/Region Level **Parameters:** *(CSL_Edma3CmdIntr*)*
CSL_EDMA3_CMD_INTR_ENABLE	Enables specified interrupts (BitMask) at Module/Region Level **Parameters:** *(CSL_Edma3CmdIntr*)*
CSL_EDMA3_CMD_INTR_DISABLE	Disables specified interrupts (BitMask) at Module/Region Level **Parameters:** *(CSL_Edma3CmdIntr*)*

CSL_EDMA3_CMD_INTR_EVAL	Interrupt Evaluation asserted for the Module/Region **Parameters:** *(Int*)*
CSL_EDMA3_CMD_CTRLERROR_CLEAR	Clear Controller Error Fault **Parameters:** *(CSL_Edma3CtrlErrStat*)*
CSL_EDMA3_CMD_EVENTMISSED_CLEAR	Pointer to an array of 3 elements, where element0 refers to the EMR register to be cleared, element1 refers to the EMRH register to be cleared, element2 refers to the QEMR register to be cleared. **Parameters:** *(CSL_BitMask32*)*
CSL_EDMA3_CMD_MEMPROTECT_SET	Programming of MPPAG,MPPA[0-7] attributes. **Parameters:** (CSL_Edma3CmdRegion*)
CSL_EDMA3_CMD_MEMFAULT_CLEAR	Clear Memory Fault **Parameters:** *(None)*

4.4.4 CSL_Edma3HwStatusQuery

enum CSL_Edma3HwStatusQuery
MODULE Level Queries.

Enumeration values:

CSL_EDMA3_QUERY_CTRLERROR	Returns Controller Error **Parameters:** *(CSL_Edma3CtrlErrStat*)*
CSL_EDMA3_QUERY_INTRPEND	Returns pend status of specified interrupt **Parameters:** *(CSL_Edma3CmdIntr*)*
CSL_EDMA3_QUERY_EVENTMISSED	Returns missed status of all Channels Pointer to an array of 3 elements, where element0 refers to the EMR register, element1 refers to the EMRH register, element2 refers to the QEMR register **Parameters:** *(CSL_BitMask32*)*
CSL_EDMA3_QUERY_QUESTATUS	Returns the Que status **Parameters:** *(CSL_Edma3QueStat*)*
CSL_EDMA3_QUERY_ACTIVITY	Returns the Channel Controller Active Status **Parameters:** *(CSL_Edma3ActivityStat*)*
CSL_EDMA3_QUERY_INFO	Returns the Channel Controller Information viz. Configuration, Revision Id **Parameters:**

	*(CSL_Edma3QueryInfo *)*
CSL_EDMA3_QUERY_MEMFAULT	Return the Memory fault details **Parameters:** *(CSL_Edma3MemFaultStat *)*
CSL_EDMA3_QUERY_MEMPROTECT	Return memory attribute of the specified region **Parameters:** *(CSL_Edma3CmdRegion *)*

4.4.5 CSL_Edma3HwChannelControlCmd

enum CSL_Edma3HwChannelControlCmd
CHANNEL Commands.

Enumeration values:

CSL_EDMA3_CMD_CHANNEL_ENABLE	Enables specified Channel **Parameters:** *(None)*
CSL_EDMA3_CMD_CHANNEL_DISABLE	Disables specified Channel **Parameters:** *(None)*
CSL_EDMA3_CMD_CHANNEL_SET	Manually sets the channel event, writes into ESR/ESRH and not ER.NA for QDMA. **Parameters:** *(None)*
CSL_EDMA3_CMD_CHANNEL_CLEAR	Manually clears the channel event, does not write into ESR/ESRH or ER/ERH but the ECR/ECRH. NA for QDMA. **Parameters:** *(None)*
CSL_EDMA3_CMD_CHANNEL_CLEARERR	In case of DMA channels clears SER/SERH (by writing into SECR/SECRH if "secEvt" and "missed" are both TRUE) and EMR/EMRH (by writing into EMCR/EMCRH if "missed" is TRUE). In case of QDMA channels clears QSER (by writing into QSECR if "ser" and "missed" are both TRUE) and QEMR (by writing into QEMCR if "missed" is TRUE) **Parameters:** *(CSL_Edma3ChannelErr *)*

4.4.6 CSL_Edma3HwChannelStatusQuery

enum CSL_Edma3HwChannelStatusQuery
CHANNEL Queries.

Enumeration values:

CSL_EDMA3_QUERY_CHANNEL_STATUS	In case of DMA channels returns TRUE if ER/ERH is set, In case of QDMA channels

	returns TRUE if QER is set **Parameters:** *(Bool*)*
CSL_EDMA3_QUERY_CHANNEL_ERR	In case of DMA channels, 'missed' is set to TRUE if EMR/EMRH is set, 'secEvt' is set to TRUE if SER/SERH is set. In case of QDMA channels, 'missed' is set to TRUE if QEMR is set, 'secEvt' is set to TRUE if QSER is set. It should be noted that if secEvt ONLY is set to TRUE it may not be a valid error condition **Parameters:** *(CSL_Edma3ChannelErr*)*

4.5 Macros

#define CSL_EDMA3_ADDRMODE_FIFO 1
Address Mode is such it wraps around after reaching FIFO width

#define CSL_EDMA3_ADDRMODE_INCR 0
Address Mode is incremental

**#define CSL_EDMA3_BIDX_MAKE (src, dst) **
```
(Uint32)(\
     CSL_FMK(EDMA3CC_SRC_DST_BIDX_DSTBIDX,(Uint32)dst) | \
     CSL_FMK(EDMA3CC_SRC_DST_BIDX_SRCBIDX,(Uint32)src)\
)
```
Used for creating the B index entry in the parameter RAM (PaRAM)

**#define CSL_EDMA3_CIDX_MAKE (src, dst) **
```
(Uint32)(\
    CSL_FMK(EDMA3CC_SRC_DST_CIDX_DSTCIDX,(Uint32)dst) | \
    CSL_FMK(EDMA3CC_SRC_DST_CIDX_SRCCIDX,(Uint32)src)\
    )
```
Used for creating the C index entry in the parameter RAM (PaRAM)

**#define CSL_EDMA3_CNT_MAKE (aCnt, bCnt) **
```
(Uint32)(\
     CSL_FMK(EDMA3CC_A_B_CNT_ACNT, aCnt) | \
     CSL_FMK(EDMA3CC_A_B_CNT_BCNT, bCnt)\
    )
```
Used for creating the A, B count entry in the parameter RAM (PaRAM)

**#define CSL_EDMA3_DMACHANNELSETUP_DEFAULT **
```
{        \
   {CSL_EDMA3_QUE_0,0}, \
   {CSL_EDMA3_QUE_0,1}, \
   {CSL_EDMA3_QUE_0,2}, \
   {CSL_EDMA3_QUE_0,3}, \
   {CSL_EDMA3_QUE_0,4}, \
   {CSL_EDMA3_QUE_0,5}, \
   {CSL_EDMA3_QUE_0,6}, \
   {CSL_EDMA3_QUE_0,7}, \
   {CSL_EDMA3_QUE_0,8}, \
   {CSL_EDMA3_QUE_0,9}, \
   {CSL_EDMA3_QUE_0,10}, \
   {CSL_EDMA3_QUE_0,11}, \
   {CSL_EDMA3_QUE_0,12}, \
   {CSL_EDMA3_QUE_0,13}, \
   {CSL_EDMA3_QUE_0,14}, \
   {CSL_EDMA3_QUE_0,15}, \
   {CSL_EDMA3_QUE_0,16}, \
   {CSL_EDMA3_QUE_0,17}, \
   {CSL_EDMA3_QUE_0,18}, \
   {CSL_EDMA3_QUE_0,19}, \
   {CSL_EDMA3_QUE_0,20}, \
   {CSL_EDMA3_QUE_0,21}, \
   {CSL_EDMA3_QUE_0,22}, \
```

```
    {CSL_EDMA3_QUE_0,23}, \
    {CSL_EDMA3_QUE_0,24}, \
    {CSL_EDMA3_QUE_0,25}, \
    {CSL_EDMA3_QUE_0,26}, \
    {CSL_EDMA3_QUE_0,27}, \
    {CSL_EDMA3_QUE_0,28}, \
    {CSL_EDMA3_QUE_0,29}, \
    {CSL_EDMA3_QUE_0,30}, \
    {CSL_EDMA3_QUE_0,31}, \
    {CSL_EDMA3_QUE_0,32}, \
    {CSL_EDMA3_QUE_0,33}, \
    {CSL_EDMA3_QUE_0,34}, \
    {CSL_EDMA3_QUE_0,35}, \
    {CSL_EDMA3_QUE_0,36}, \
    {CSL_EDMA3_QUE_0,37}, \
    {CSL_EDMA3_QUE_0,38}, \
    {CSL_EDMA3_QUE_0,39}, \
    {CSL_EDMA3_QUE_0,40}, \
    {CSL_EDMA3_QUE_0,41}, \
    {CSL_EDMA3_QUE_0,42}, \
    {CSL_EDMA3_QUE_0,43}, \
    {CSL_EDMA3_QUE_0,44}, \
    {CSL_EDMA3_QUE_0,45}, \
    {CSL_EDMA3_QUE_0,46}, \
    {CSL_EDMA3_QUE_0,47}, \
    {CSL_EDMA3_QUE_0,48}, \
    {CSL_EDMA3_QUE_0,49}, \
    {CSL_EDMA3_QUE_0,50}, \
    {CSL_EDMA3_QUE_0,51}, \
    {CSL_EDMA3_QUE_0,52}, \
    {CSL_EDMA3_QUE_0,53}, \
    {CSL_EDMA3_QUE_0,54}, \
    {CSL_EDMA3_QUE_0,55}, \
    {CSL_EDMA3_QUE_0,56}, \
    {CSL_EDMA3_QUE_0,57}, \
    {CSL_EDMA3_QUE_0,58}, \
    {CSL_EDMA3_QUE_0,59}, \
    {CSL_EDMA3_QUE_0,60}, \
    {CSL_EDMA3_QUE_0,61}, \
    {CSL_EDMA3_QUE_0,62}, \
    {CSL_EDMA3_QUE_0,63} \
}
```

DMA Channel Setup

#define CSL_EDMA3_FIFOWIDTH_NONE 0
Only for ease

#define CSL_EDMA3_FIFOWIDTH_8BIT 0
8 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_16BIT 1
16 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_32BIT 2
32 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_64BIT 3
64 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_128BIT 4
128 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_256BIT 5
256 bit FIFO Width

#define CSL_EDMA3_ITCCH_DIS 0
Intermediate transfer completion chaining disable

#define CSL_EDMA3_ITCCH_EN 1
Intermediate transfer completion chaining enable

#define CSL_EDMA3_ITCINT_DIS 0
Intermediate transfer completion interrupt disable

#define CSL_EDMA3_ITCINT_EN 1
Intermediate transfer completion interrupt enable

#define CSL_EDMA3_LINK_DEFAULT 0xFFFF
Link to a Null PaRAM set

#define CSL_EDMA3_LINK_NULL 0xFFFF
Link to a Null PaRAM set

**#define CSL_EDMA3_LINKBCNTRLD_MAKE (link, bCntRld) **

```
(Uint32)(\
     CSL_FMK(EDMA3CC_LINK_BCNTRLD_LINK,(Uint32)link) | \
     CSL_FMK(EDMA3CC_LINK_BCNTRLD_BCNTRLD,bCntRld) \
  )
```

Used for creating the link and B count reload entry in the parameter RAM (PaRAM)

**#define CSL_EDMA3_OPT_MAKE (itcchEn, tcchEn, itcintEn, tcintEn, tcc, tccMode, **
**fwid, stat, syncDim, dam, sam) **

```
(Uint32)( \
         CSL_FMKR(23,23,itcchEn) | \
         CSL_FMKR(22,22,tcchEn) | \
         CSL_FMKR(21,21,itcintEn) | \
         CSL_FMKR(20,20,tcintEn) | \
         CSL_FMKR(17,12,tcc) | \
         CSL_FMKR(11,11,tccMode) | \
         CSL_FMKR(10,8,fwid) | \
         CSL_FMKR(3,3,stat) | \
         CSL_FMKR(2,2,syncDim) | \
         CSL_FMKR(1,1,dam) | \
         CSL_FMKR(0,0,sam))
```

Used for creating the options entry in the parameter RAM

**#define CSL_EDMA3_QDMACHANNELSETUP_DEFAULT **

```
{       \
   {CSL_EDMA3_QUE_0,64,CSL_EDMA3_TRIGWORD_DEFAULT}, \
   {CSL_EDMA3_QUE_0,65,CSL_EDMA3_TRIGWORD_DEFAULT}, \
   {CSL_EDMA3_QUE_0,66,CSL_EDMA3_TRIGWORD_DEFAULT}, \
   {CSL_EDMA3_QUE_0,67,CSL_EDMA3_TRIGWORD_DEFAULT}  \
}
```

QDMA Channel Setup

#define CSL_EDMA3_STATIC_DIS 0
Disable Static

#define CSL_EDMA3_STATIC_EN 1
Enable Static

#define CSL_EDMA3_SYNC_A 0
A synchronized transfer

#define CSL_EDMA3_SYNC_AB 1
AB synchronized transfer

#define CSL_EDMA3_TCC_EARLY 1
Early Completion

#define CSL_EDMA3_TCC_NORMAL 0
Normal Completion

#define CSL_EDMA3_TCCH_DIS 0
Transfer completion chaining disable

#define CSL_EDMA3_TCCH_EN 1
Transfer completion chaining enable

#define CSL_EDMA3_TCINT_DIS 0
Transfer completion interrupt disable

#define CSL_EDMA3_TCINT_EN 1
Transfer completion interrupt enable

#define CSL_EDMA3_TRIGWORD_DEFAULT 7
Last trigger word in a QDMA parameter RAM set

/** Trigger word option field */
#define CSL_EDMA3_TRIGWORD_OPT 0

/** Trigger word source */
#define CSL_EDMA3_TRIGWORD_SRC 1

/** Trigger word AB count */
#define CSL_EDMA3_TRIGWORD_A_B_CNT 2

/** Trigger word destination */
#define CSL_EDMA3_TRIGWORD_DST 3

/** Trigger word src and dst B index */
#define CSL_EDMA3_TRIGWORD_SRC_DST_BIDX 4

/** Trigger word B count reload */
#define CSL_EDMA3_TRIGWORD_LINK_BCNTRLD 5

/** Trigger word src and dst C index */
#define CSL_EDMA3_TRIGWORD_SRC_DST_CIDX 6

/** Trigger word C count */
#define CSL_EDMA3_TRIGWORD_CCNT 7

#define CSL_EDMA3_MEMACCESS_UX 0x0001
User Execute permission

#define CSL_EDMA3_MEMACCESS_UW 0x0002
User Write permission

#define CSL_EDMA3_MEMACCESS_UR 0x0004
User Read permission

#define CSL_EDMA3_MEMACCESS_SX 0x0008
Supervisor Execute permission

#define CSL_EDMA3_MEMACCESS_SW 0x0010
Supervisor Write permission

#define CSL_EDMA3_MEMACCESS_SR 0x0020
Supervisor Read permission

#define CSL_EDMA3_MEMACCESS_EXT 0x0200
External Allowed ID. Requests with PrivID >= '6' are permitted if access type is allowed

#define CSL_EDMA3_MEMACCESS_AID0 0x0400
Allowed ID '0'

#define CSL_EDMA3_MEMACCESS_AID1 0x0800
Allowed ID '1'

#define CSL_EDMA3_MEMACCESS_AID2 0x1000
Allowed ID '2'

#define CSL_EDMA3_MEMACCESS_AID3 0x2000
Allowed ID '3'

#define CSL_EDMA3_MEMACCESS_AID4 0x4000
Allowed ID '4'

#define CSL_EDMA3_MEMACCESS_AID5 0x8000
Allowed ID '5'

4.6 Typedefs

typedef void * CSL_Edma3Context
Module specific context information. This is a dummy handle.

typedef void * CSL_Edma3ModuleAttr
Module Attributes specific information. This is a dummy handle.

typedef volatile CSL_Edma3ccParamsetRegs * CSL_Edma3ParamHandle
CSL Parameter RAM (PaRAM) Set Handle.

typedef CSL_Edma3Obj * CSL_Edma3Handle
EDMA handle.

typedef CSL_Edma3ChannelObj * CSL_Edma3ChannelHandle
CSL Channel Handle All channel level API calls must be made with this handle.

Chapter 5
EDMA2.x Module

Topics

5.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within EDMA2.x (backward compatible with EDMA csl2x APIs) module.

The EDMA controller handles all data transfers between the level-two (L2) cache/memory controller and the device peripherals.These data transfers include cache servicing, non-cacheable memory accesses, user-programmed data transfers, and host accesses. The EDMA supports up to 64-event channels and 4 QDMA channels. The EDMA consists of a scalable Parameter RAM (PaRAM) that supports flexible ping-pong, circular buffering, channel-chaining, auto-reloading, and memory protection. The EDMA allows movement of data to/from any addressable memory spaces, including internal memory (L2 SRAM), peripherals, and external memory.

5.2 Functions

This section lists the functions available in the EDMA2 module.

5.2.1 EDMA_reset

CSLAPI void EDMA_reset (EDMA_Handle *hEdma*)

Description
This function resets the given EDMA channel.

Arguments

```
hEdma      Handle to the channel to be reset
```

Return Value
None

Pre Condition
Channel must have been opened previously using the function *EDMA_open()*.

Post Condition

- The corresponding PaRAM entry is cleared to 0.
- The channel is disabled and event register bit is cleared.

Modifies
The system data structures are modified.

Example

```
EDMA_Handle handle;
Uint32      chan_no = 1;

handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_reset(handle);
...
```

5.2.2 EDMA_resetAll

CSLAPI void EDMA_resetAll (void)

Description
This function resets all EDMA channels.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition

- The PaRAM words corresponding to all of the EDMA channels are cleared to 0.
- All channels are disabled and their interrupts are reset.

Modifies
The system data structures are modified.

Example
```
...
EDMA_resetAll();
...
```

5.2.3 EDMA_open

CSLAPI EDMA_Handle EDMA_open (int *channel*, Uint32 *flags*)

Description
This function opens a EDMA channel for use by the application.

Arguments

```
channel     Channel number or EDMA_CHA_ANY (to open any channel)

flags       EDMA_OPEN_RESET or EDMA_OPEN_ENABLE or 0
            EDMA_OPEN_RESET  - resets the channel
            EDMA_OPEN_ENABLE - enables the transfers
```

Return Value

- A valid handle on success
- EDMA_HINV on failure

Pre Condition
None

Post Condition
The channel is enabled or reset (PaRAM entry is cleared and channel is disabled) depending on the flags passed.

Modifies
The system data structures are modified.

Example
```
Uint32             chan_no = 4;
EDMA_Handle        handle;
handle = EDMA_open(chan_no, 0);
```

5.2.4 EDMA_close

CSLAPI void EDMA_close (EDMA_Handle *hEdma*)

Description
Closes a previously opened EDMA channel, after it has been used by the application.

Arguments

```
hEdma      Handle to the channel to be closed
```

Return Value
None

Pre Condition
The channel to be closed must have been opened previously using the function EDMA_open ().

Post Condition
The channel is closed and reset.

Modifies
The system data structures are modified.

Example

```
EDMA_Handle  handle;
Uint32       chan_no = 1;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_close(handle);
...
```

5.2.5 EDMA_allocTable

CSLAPI EDMA_Handle EDMA_allocTable (int *tableNum*)

Description
This function allocates a PaRAM table entry for use by the application.

Arguments

```
tableNum         PaRAM table entry number (0 to EDMA_TABLE_CNT)
or
EDMA_ALLOC_ANY   To allocate any available entry of PaRAM table
```

Return Value

- A valid handle on success
- EDMA_HINV on failure

Pre Condition
None

Post Condition
A PaRAM table entry is allocated from the free pool.

Modifies
The system data structures are modified.

Example

```
EDMA_Handle        handle;
Uint32             tabNum = 1;
...
handle = EDMA_allocTable(tabNum);
...
```

5.2.6 EDMA_freeTable

CSLAPI void EDMA_freeTable (EDMA_Handle *hEdma*)

Description
This function frees a previously allocated PaRAM table entry.

Arguments

```
hEdma      Handle to the PaRAM entry to be freed
```

Return Value
None

Pre Condition
The table entry to be freed must have been allocated previously using function EDMA_allocTable().

Post Condition
One more entry in the free PaRAM table.

Modifies
The system data structures are modified.

Example

```
EDMA_Handle  handle;
Uint32       tabNum = 1;
...
handle = EDMA_allocTable(tabNum);
EDMA_freeTable(handle);
...
```

5.2.7 EDMA_allocTableEx

CSLAPI int EDMA_allocTableEx (int *cnt*, EDMA_Handle* *array*)

Description
This function allocates a number of PaRAM table entries from the free pool.

Arguments

```
cnt        Number of channels to be allocated

array      Pointer to the first element of array of EDMA handles
           to return handles for the allocated entries
```

Return Value

- The number of allocated entries, if success
- 0, if failure

Pre Condition
None

Post Condition
The number of entries in free PaRAM table is less by 'cnt'

Modifies
The system data structures are modified.

Example

```
EDMA_Handle  hArray[4];
Uint32       cnt = 4, retCnt;
...
retCnt = EDMA_allocTableEx(cnt, &hArray[0]);
...
```

5.2.8 EDMA_freeTableEx

CSLAPI void EDMA_freeTableEx	**(**	**int**	***cnt*,**
		EDMA_Handle*	***array***
	)		

Description
This function frees previously allocated PaRAM table entries.

Arguments

```
cnt        Number of channels to be freed

array      Pointer to the first element of array of EDMA handles
           that are to be freed
```

Return Value
None

Pre Condition
To be freed entries must have been allocated previously using function EDMA_allocTableEx().

Post Condition
The number of entries in free PaRAM table is more by 'cnt'

Modifies
The system data structures are modified.

Example

```
EDMA_Handle       hArray[4];
Uint32            cnt = 4, retCnt;
retCnt = EDMA_allocTableEx(cnt, &hArray[0]);
...
EDMA_freeTableEx(cnt, &hArray[0]);
...
```

5.2.9 EDMA_clearPram

CSLAPI void EDMA_clearPram (Uint32 *val*)

Description
This function sets the PARAM words corresponding to all EDMA channels with the specified 'val'.

Arguments

```
val      Value to be written into the PaRAM words
```

Return Value
None

Pre Condition
None

Post Condition
All words of the PaRAM corresponding to the EDMA channels are set to the given value, 'val'. Reserved fields of PaRAM do not reflect the written bit values. They are read as zero.

Modifies
None

Example

```
Uint32 val = 0;
...
EDMA_clearPram(val);
...
```

5.2.10 EDMA_intAlloc

CSLAPI int EDMA_intAlloc (int *tcc*)

Description
This function allocates a EDMA channel interrupt. This interrupt is used in channel configuration to configure the interrupt to be generated after a transfer.

Arguments

```
tcc      Interrupt number
or
```

```
-1        To allocate any available interrupt
```

Return Value

- Interrupt number allocated, if success
- -1, to indicate failure

Pre Condition
None

Post Condition
One interrupt less in the free pool of interrupts.

Modifies
The system data structures are modified.

Example

```
Uint32 tcc = 1, retTcc;
 ...
retTcc = EDMA_intAlloc(tcc);
 ...
```

5.2.11 EDMA_intFree

CSLAPI void EDMA_intFree (int *tcc*)

Description
This function frees a previously allocated interrupt.

Arguments

```
tcc      Interrupt number to be freed
```

Return Value
None

Pre Condition
The EDMA_intAlloc() function must be called to allocate the interrupt before calling this function.

Post Condition
One interrupt more in the free pool.

Modifies
The system data structures are modified.

Example

```
Uint32 tcc = 1, retTcc;
 ...
retTcc = EDMA_intAlloc(tcc);
 ...
EDMA_intFree(retTcc);
 ...
```

5.2.12 EDMA_config

CSLAPI void EDMA_config (EDMA_Handle *hEdma*, EDMA_Config * *config*)

Description
This function configures an EDMA transfer.

1. Following transfers specified in the document 'spru234.pdf' are NOT possible. When these are configured in the EDMA_Config structure, the routine returns without configuring the PaRAM.

As per the document, the following transfers are NOT possible :
A-44, A-47, A-48, A-49, A-50, A-62, A-65, A-66, A-67,
A-68, A-80, A-83 A-84, A-85 and A-86.

All these "NOT POSSIBLE" are possible by setting ACnt = elmSize, BCnt = elmCnt, CCnt =arCnt+1, appropriate indexes and these chain to themselves. But TCC = channel Number should be free and this programmation should not contrast with user programmation of Interrupt enables/chain enables/tcc programmation.

2. For the following transfers specified in the document 'spru234.pdf', the source address must be aligned to 256-bits, otherwise the config API returns without configuring.
A-42, A-43, A-60, A-61 and A-78

3. For the following transfers specified in the document 'spru234.pdf', the destination address must be aligned to 256-bits, otherwise the config API returns without configuring.
A-42, A-45, A-60, A-63, A-66, A78 and A-81

Arguments

`hEdma`	`Handle to the channel or PaRAM to be configured`
`config`	`Address of the configuration structure`

Return Value
None

Pre Condition
1. Channel must have been opened or a PaRAM entry allocated.
2. A TCC must have been allocated, if TCINT bit is set.

Post Condition
The corresponding PaRAM entry is configured, if the configuration is valid.

Modifies
The PaRAM is modified if the configuration is valid.

Example
```
EDMA_Handle handle;
Uint32      chan_no = 1, tcc;
EDMA_Config conf;
char        dst[512];
```

```
    char          src[512];

    handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
    tcc = EDMA_intAlloc(4);
    conf.opt = 0x51340001;
    conf.cnt = 0x00000200;  /* Transfer 512 bytes */
    conf.idx = 0;
    conf.rld = 0x0000FFFF;
    conf.dst = (Uint32)&dst[0];
    conf.src = (Uint32)&src[0];
    ...
    EDMA_config(handle, &conf);
    ...
```

5.2.13 EDMA_configArgs

CSLAPI void EDMA_configArgs	**(**	**EDMA_Handle**	***hEdma*,**
		Uint32	***opt*,**
		Uint32	***src*,**
		Uint32	***cnt*,**
		Uint32	***dst*,**
		Uint32	***idx*,**
		Uint32	***rld***
	)		

Description
This function configures an EDMA transfer.

1. Following transfers specified in the document 'spru234.pdf' are NOT possible. When these are configured in the EDMA_Config structure, the routine returns without configuring the PaRAM.

As per the document, the following transfers are NOT possible:
A-44, A-47, A-48, A-49, A-50, A-62, A-65, A-66, A-67,
A-68, A-80, A-83 A-84, A-85 and A-86.

All these "NOT POSSIBLE" are possible by setting ACnt = elmSize, BCnt = elmCnt, CCnt =arCnt+1, appropriate indexes and these chain to themselves. But TCC = channel Number should be free and this programmation should not contrast with user programmation of Interrupt enables/chain enables/tcc programmation.

2. For the following transfers specified in the document 'spru234.pdf', the source address must be aligned to 256-bits, otherwise the config API returns without configuring.
A-42, A-43, A-60, A-61 and A-78

3. For the following transfers specified in the document 'spru234.pdf', the destination address must be aligned to 256-bits, otherwise the config API returns without configuring.
A-42, A-45, A-60, A-63, A-66, A78 and A-81

Arguments

```
hEdma      Handle to the channel or PaRAM to be configured

opt        Options word of the configuration

src        From address used in the transfer

cnt        Specify the number of arrays and number of elements
           in each array

dst        To address used in the transfer

idx        Specify offsets used to calculate the addresses

rld        Specify the link address and the reload value
```

Return Value
None

Pre Condition
1. Channel must have been opened or a PaRAM entry allocated.
2. A TCC must have been allocated, if TCINT bit is set.

Post Condition
The corresponding PaRAM entry is configured, if the configuration is valid.

Modifies
The PaRAM is modified if the configuration is valid.

Example

```
EDMA_Handle handle;
Uint32      chan_no = 1, tcc;
Uint32      opt, cnt, idx, rld, src, dst;
char        dst1[512];
char        src1[512];

handle = EDMA_open(chan_no, EDMA_OPEN_RESET);

tcc = EDMA_intAlloc(4);
...
opt = 0x51340001;
cnt = 0x00000200;  /* Transfer 512 bytes */
idx = 0;
rld = 0x0000FFFF;
dst = (Uint32)&dst1[0];
src = (Uint32)&src1[0];

EDMA_configArgs(handle, opt, src, cnt, dst, idx, rld);
...
```

5.2.14 EDMA_getConfig

CSLAPI void EDMA_getConfig **(** [EDMA_Handle] ***hEdma*,** [EDMA_Config] ***** ***config*** **)**

Description
This function returns the configuration of an EDMA transfer, with the following limitations.

> Fields - 2DS, SUM, 2DD, DUM, PDTS, PDTD, FS, FRMIDX, ELEIDX and ELERLD are not returned (not modified in the argument structure passed to the API).
> Fields - ATCINT, ATCC, LINK are valid if the programmed values are validOther fields contain valid configuration.

Arguments

`hEdma`	`Handle of the channel`
`config`	`Pointer to the configuration structure of type 'EDMA_Config'`

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
EDMA_Handle     handle;
Uint32          chan_no = 1;
EDMA_Config     conf;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_getConfig(handle, &conf);
...
```

5.2.15 EDMA_qdmaConfig

CSLAPI void EDMA_qdmaConfig **(** [EDMA_Config] ***** ***config*** **)**

Description
This function configures a QDMA transfer, returns after initiating the transfer.

1. Following transfers specified in the document, 'spru234.pdf' are NOT possible. When these are configured in the EDMA_Config structure, the routine returns without a data transfer.

   The following transfers are NOT possible :
   A-44, A-47, A-48, A-49, A-50, A-62, A-65, A-66, A-67,
   A-68, A-80, A-83 A-84, A-85 and A-86.

2. For the following transfers specified in the document 'spru234.pdf', the source address must be aligned to 256-bits, otherwise the config API returns without a data transfer. A-42, A-43, A-60, A-61 and A-78

3. For the following transfers specified in the document 'spru234.pdf', the destination address must be aligned to 256-bits, otherwise the config API returns without a data transfer.
   A-42, A-45, A-60, A-63, A-66, A78 and A-81

4. No need to enable the QDMA channel separately, this API takes care of enabling the QDMA channel.

5. All transfers with QDMA are frame-synchronized transfers.

6. Only one QDMA channel supported; Linking and Chaining are not supported.

Arguments

```
config       Address of the configuration structure
```

Return Value
None

Pre Condition
A TCC must have been allocated, if TCINT bit is set.

Post Condition
The corresponding PaRAM entry is configured, if the configuration is valid.

Modifies
The PaRAM is modified if the configuration is valid.

Example

```
EDMA_Config conf;
EDMA_Handle handle;
Uint32      chan_no = 64;
Uint32      tcc;
char        dst[512];
char        src[512];

handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
tcc = EDMA_intAlloc(4);
...
conf.opt = 0x51340001;
conf.cnt = 0x00000200;  /* Transfer 512 bytes*/
conf.idx = 0;
conf.dst = (Uint32)&dst[0];
```

```
conf.src = (Uint32)&src[0];
EDMA_qdmaConfig(&conf);
...
```

5.2.16 EDMA_qdmaConfigArgs

CSLAPI void EDMA_qdmaConfigArgs (Uint32 *opt*, Uint32 *src*, Uint32 *cnt*, Uint32 *dst*, Uint32 *idx*)

Description
This function configures a QDMA transfer, returns after initiating the transfer.

1. Following transfers specified in the document 'spru234.pdf' are NOT possible. When these are configured in the EDMA_Config structure, the routine returns without a data transfer.

The following transfers are NOT possible:
A-44, A-47, A-48, A-49, A-50, A-62, A-65, A-66, A-67,
A-68, A-80, A-83 A-84, A-85 and A-86.

2. For the following transfers specified in the document 'spru234.pdf', the source address must be aligned to 256-bits, otherwise the config API returns without a data transfer.
A-42, A-43, A-60, A-61 and A-78

3. For the following transfers specified in the document 'spru234.pdf', the destination address must be aligned to 256-bits, otherwise the config API returns without a data transfer.
A-42, A-45, A-60, A-63, A-66, A78 and A-81

4. No need to enable the QDMA channel separately, this API takes care of enabling the QDMA channel.

5. All transfers with QDMA are frame-synchronized transfers.

6. Only one channel of QDMA is supported, Linking and Chaining are not supported.

Arguments

```
opt         Options word of the configuration

src         From address used in the transfer

cnt         Specify the number of arrays and number of elements
            in each array

dst         To address used in the transfer
```

```
idx         Specify offsets used to calculate the addresses
```

Return Value
None

Pre Condition
A TCC must have been allocated, if TCINT bit is set.

Post Condition
The corresponding PaRAM entry is configured, if the configuration is valid.

Modifies
The PaRAM is modified if the configuration is valid.

Example

```
Uint32        opt, cnt, idx, src, dst, tcc, rld;
char          dst1[512];
char          src1[512];
 ...
tcc = EDMA_intAlloc(4);
opt = 0x51340001;
cnt = 0x00000200;  /* Transfer 512 bytes*/
idx = 0;
rld = 0x0000FFFF;
dst = (Uint32)&dst1[0];
src = (Uint32)&src1[0];

EDMA_qdmaConfigArgs(opt, src, cnt, dst, idx);
...
```

5.2.17 EDMA_qdmaGetConfig

CSLAPI void EDMA_qdmaGetConfig (EDMA_Config * *config*)

Description
This function returns the configuration of a QDMA transfer, with the following limitations.

Fields - ESIZE, 2DS, SUM, 2DD, DUM, PDTS, PDTD, FRMCNT, ELECNT, FRMIDX and ELEIDX are not returned (not modified in the argument structure passed to the API).
Fields - FS returned as 1, reserved fields are DO NOT CARE. Other fields contain valid configuration.

Arguments

```
config       A pointer to EDMA_Config structure to return the
             configuration
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
EDMA_Config conf;
EDMA_Handle handle;
Uint32      chan_no = 64;

handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_qdmaGetConfig(&conf);
...
```

5.2.18 EDMA_getScratchAddr

IDECL Uint32 EDMA_getScratchAddr (void)

Description
This function returns the address of the scratch area. Some portion of PaRAM area is reserved for scratch purposes.

Arguments
None

Return Value
Address of the scratch area

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 *addr;
...
addr = (Uint32 *)EDMA_getScratchAddr();
...
```

5.2.19 EDMA_getScratchSize

IDECL Uint32 EDMA_getScratchSize (void)

Description
This function returns the size of scratch area in bytes.

Arguments
None

Return Value
Scratch area size in bytes

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 scratchSize;
...
scratchSize = EDMA_getScratchSize();
...
```

5.2.20 EDMA_enableChannel

IDECL void EDMA_enableChannel (EDMA_Handle *hEdma*)

Description
This function enables a channel for used by a peripheral / host.

Arguments

```
hEdma      Handle to the channel to be enabled
```

Return Value
None

Pre Condition
Channel must have been opened using EDMA_open() before calling this function..

Post Condition
The corresponding channel is ready for use by the peripheral.

Modifies
Sets a bit in EER or EERH.

Example

```
EDMA_Handle       handle;
Uint32            chan_no = 1;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_enableChannel(handle);
...
```

5.2.21 EDMA_disableChannel

IDECL void EDMA_disableChannel (EDMA_Handle *hEdma*)

Description
This function disables a channel after its use by a peripheral / host.

Arguments

```
hEdma      Handle to the channel to be disabled
```

Return Value
None

Pre Condition
Channel must have been opened and enabled before calling this function..

Post Condition
The corresponding channel is no longer usable by the peripheral.

Modifies
Clears a bit in EER or EERH.

Example

```
EDMA_Handle  handle;
Uint32       chan_no = 1;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_enableChannel(handle);
EDMA_disableChannel(handle);
...
```

5.2.22 EDMA_setChannel

IDECL void EDMA_setChannel (EDMA_Handle *hEdma*)

Description
This function initiates a transfer on a channel.

Arguments

```
hEdma      Handle to the channel to be triggered
```

Return Value
None

Pre Condition
Channel must have been opened and configured before calling this fucntion.

Post Condition
Starts the transfer configured for the channel.

Modifies
Sets a bit in ER or ERH.

Example
```
EDMA_Handle  handle;
Uint32       chan_no = 1;
EDMA_Config  conf;
char         dst[512];
char         src[512];

handle = EDMA_open(chan_no, EDMA_OPEN_RESET);

conf.opt = 0x51340001;
conf.cnt = 0x00000200;  /* Transfer 512 bytes*/
conf.idx = 0;
conf.rld = 0x0000FFFF;
conf.dst = (Uint32)&dst[0];
conf.src = (Uint32)&src[0];
...


EDMA_config(handle, &conf);
EDMA_setChannel(handle);
...
```

5.2.23 EDMA_getChannel

IDECL Uint32 EDMA_getChannel (EDMA_Handle *hEdma*)

Description
This function returns the current state of the channel event by reading the event flag from the EDMA channel Event Register (ER).

Arguments

```
hEdma      Handle to the channel to be tested
```

Return Value

- 0 - event not detected
- 1 - event detected

Pre Condition
Channel must have been opened and enabled before calling this function.

Post Condition
None

Modifies
None

Example
```
EDMA_Handle        handle;
Uint32             chan_no = 1, status;
```

```
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_enableChannel(handle);
...
status = EDMA_getChannel(handle);
...
```

5.2.24 EDMA_clearChannel

IDECL void EDMA_clearChannel (EDMA_Handle *hEdma*)

Description
This function clears a peripheral transfer request on a channel.

Arguments

```
hEdma      Handle to the channel to be cleared
```

Return Value
None

Pre Condition
Channel must have been opened before calling this function.

Post Condition
A bit in the event register is cleared. This stops EDMA from transferring data if transfer has not been started.

Modifies
Clears a bit in ER or ERH.

Example

```
EDMA_Handle  handle;
Uint32       chan_no = 1;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_clearChannel(handle);
...
```

5.2.25 EDMA_getTableAddress

IDECL Uint32 EDMA_getTableAddress (EDMA_Handle *hEdma*)

Description
This function returns address of PaRAM corresponding to the given EDMA handle.

Arguments

```
hEdma      Handle to the channel or table entry
```

Return Value
Address of the corresponding PaRAM entry

Pre Condition
Channel must have been opened or a table entry allocated before calling this function.

Post Condition
None

Modifies
None

Example

```
EDMA_Handle  handle;
Uint32       chan_no = 1, *addr;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
 ...
addr = (Uint32 *)EDMA_getTableAddress(handle);
...
```

5.2.26 EDMA_intEnable

IDECL void EDMA_intEnable (Uint32 *tccIntNum*)

Description
This function enables a transfer completion interrupt.

Arguments

```
tccIntNum      Interrupt number to be enabled
```

Return Value
None

Pre Condition
None

Post Condition
The future EDMA events of this number can interrupt the CPU.

Modifies
A bit in IER or IERH is set.

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1;
 ...
EDMA_intEnable(tccIntNum);
...
```

5.2.27 EDMA_intDisable

IDECL void EDMA_intDisable (Uint32 *tccIntNum*)

Description
This function disables a transfer completion interrupt.

Arguments

```
tccIntNum      Interrupt number to be disabled
```

Return Value
None

Pre Condition
None

Post Condition
The future EDMA events of this number cannot interrupt the CPU.

Modifies
A bit in IER or IERH is cleared

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1;
 ...
EDMA_intDisable(tccIntNum);
...
```

5.2.28 EDMA_intClear

IDECL void EDMA_intClear (Uint32 *tccIntNum*)

Description
This function clears a transfer completion interrupt.

Arguments

```
tccIntNum      Interrupt number to be cleared
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
A bit in IPR or IPRH is cleared.

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1;
```

```
...
EDMA_intClear(tccIntNum);
...
```

5.2.29 EDMA_intTest

IDECL Uint32 EDMA_intTest (Uint32 *tccIntNum*)

Description
This function returns the status of a transfer completion interrupt.

Arguments

```
tccIntNum      Interrupt number to be tested
```

Return Value

- 0 = flag not set
- 1 = flag set

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1, status;
 ...
status = EDMA_intTest(tccIntNum);
...
```

5.2.30 EDMA_intReset

IDECL void EDMA_intReset (Uint32 *tccIntNum*)

Description
This function clears a pending transfer completion interrupt and disables the interrupt.

Arguments

```
tccIntNum      Interrupt number to be reset
```

Return Value
None

Pre Condition
None

Post Condition
Interrupts are not recognized.

Modifies
A bit in IPR or IPRH and IER or IERH cleared.

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1, status;
 ...
EDMA_intReset(tccIntNum);
...
```

5.2.31 EDMA_intResetAll

IDECL void EDMA_intResetAll (void)

Description
This function clears all pending transfer completion interrupts and disables the all interrupts.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
Interrupts are not recognized.

Modifies
All bits in IPR, IPRH, IER and IERH cleared.

Example

```
EDMA_Handle  handle;
Uint32       tccIntNum = 1, status;
 ...
EDMA_intResetAll();
 ...
```

5.2.32 EDMA_link

IDECL void EDMA_link (EDMA_Handle *parent*, EDMA_Handle *child*)

Description
This function links a child PaRAM entry to the parent PaRAM.

Arguments

```
parent      Handle to the parent PaRAM (channel)

child       Handle to the child PaRAM
```

Return Value
None

Pre Condition
Parent and child must have been configured.

Post Condition
The parent's RLD word of PaRAM is set to the offset of child PaRAM.

Modifies
Parent PaRAM is modified.

Example

```
EDMA_Handle  par_handle, ch_handle;
Uint32       chan_no = 1, tab = 4;
par_handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
ch_handle = EDMA_allocTable(tab);

/* Configure parent, using EDMA_config */
 ...
/* Configure child, using EDMA_config */
EDMA_link(par_handle, ch_handle);
...
```

5.2.33 EDMA_chain

IDECL void EDMA_chain	**(**	EDMA_Handle	***parent*,**
		EDMA_Handle	***nextChannel*,**
		Uint32	***tccflag*,**
		Uint32	***atccflag***
	)		

Description
This function enables chaining of parent and child, after transfer gets finished/submitted based on the flags supplied.

Arguments

```
parent        EDMA handle of the channel after which next
              channel gets chained.

nextChannel   EDMA handle associated with the channel to be
              chained
```

```
tccflag       Flag to enable/disable chaining the child after
              completion of the parent transfer.
              Following are the values:
              0   -   Disable
              1   -   Enable

atccflag      Flag to enable/disable chaining the child after
              submission of  the parent transfer.
              Following are the values:
              0   -   Disable
              1   -   Enable
```

Return Value
None

Pre Condition
Channel must be opened before calling this function.

Post Condition

- Child channel gets chained to parent.
- Enables chaining of parent (child transfer gets triggered, after parent transfer gets completed), if TCC flag is set.
- Enables alternate chaining of parent (child transfer gets triggered, after parent transfer gets submitted), if ATCC flag is set.

Modifies
The OPT word of the parent PaRAM gets modified.

Example
```
EDMA_Handle  par_handle, next_handle;
Uint32       par_chan_no = 1, next_chan_no = 4;
Int atccflag = 1;

par_handle = EDMA_open(par_chan_no, EDMA_OPEN_RESET);
par_handle = EDMA_open(next_chan_no, EDMA_OPEN_RESET);

/* Configure parent using EDMA_config */
 ...
/* Configure next using EDMA_config */
...
EDMA_chain(par_handle, next_handle, 0, atccflag);
...
```

5.2.34 EDMA_enableChaining

IDECL void EDMA_enableChaining (EDMA_Handle *hEdma*)

Description
This function enables chaining on the given channel.

Arguments

```
hEdma      Handle to the channel to be chained
```

Return Value
None

Pre Condition
Channel must have been opened before calling this function.

Post Condition
The channel initiates a transfer on the channel specified in its TCC and TCCM fields, after transfer of the current channel is over.

Modifies
Associated PaRAM

Example
```
EDMA_Handle     handle;
Uint32          chan_no = 1;
handle = EDMA_open(chan_no, EDMA_OPEN_RESET);
...
EDMA_enableChaining(handle);
...
```

5.2.35 EDMA_disableChaining

IDECL void EDMA_disableChaining (EDMA_Handle *hEdma*)

Description
This function disables chaining on the given channel.

Arguments

```
hEdma      Handle to the channel whose chaining is to be disabled
```

Return Value
None

Pre Condition
Channel must have been opened before calling this function.

Post Condition
The channel does NOT initiate a transfer on the channel specified in its TCC and TCCM fields, after transfer of the current channel is over.

Modifies
Associated PaRAM

Example
```
EDMA_Handle      par_handle, next_handle;
EDMA_Config      par_conf, next_conf;
Uint32           par_chan_no = 1, next_chan_no = 4;
par_handle       = EDMA_open(par_chan_no, EDMA_OPEN_RESET);
par_handle       = EDMA_open(next_chan_no, EDMA_OPEN_RESET);
/* Configure parent using EDMA_config, with tcc field set as
next */
```

```
/* Configure next using EDMA_config */
EDMA_enableChaining(par_handle);
/* Program/peripheral initates a transfer on parent */
/* Wait for the completion of transfer on next_chan_no */
EDMA_disableChaining(par_handle);
...
```

5.3 Data Structures

This section lists the data structures available in the EDMA2 module.

5.3.1 EDMA_Config

Detailed Description
EDMA PaRAM configuration structure

Field Documentation

Uint32 EDMA_Config::cnt
Transfer count word

Uint32 EDMA_Config::dst
Destination address word

Uint32 EDMA_Config::idx
Index configuration word

Uint32 EDMA_Config::opt
Options word of the configuration

Uint32 EDMA_Config::rld
Reload address and Link offset word

Uint32 EDMA_Config::src
Source address word

5.4 Macros

#define EDMA_ALLOC_ANY (-1)
Argument used to allocate a unspecific resource of a type
Example: EDMA_open(EDMA_ALLOC_ANY, EDMA_OPEN_RESET);

#define EDMA_CHA_DSPINT 0
EDMA channel 0

#define EDMA_CHA_TINT0L 1
EDMA channel 1

#define EDMA_CHA_TINT0H 2
EDMA channel 2

#define EDMA_CHA_3 3
EDMA channel 3

#define EDMA_CHA_4 4
EDMA channel 4

#define EDMA_CHA_5 5
EDMA channel 5

#define EDMA_CHA_6 6
EDMA channel 6

#define EDMA_CHA_7 7
EDMA channel 7

#define EDMA_CHA_8 8
EDMA channel 8

#define EDMA_CHA_9 9
EDMA channel 9

#define EDMA_CHA_10 10
EDMA channel 10

#define EDMA_CHA_11 11
EDMA channel 11

#define EDMA_CHA_XEVT0 12
EDMA channel 12

#define EDMA_CHA_REVT0 13
EDMA channel 13

#define EDMA_CHA_XEVT1 14
EDMA channel 14

#define EDMA_CHA_REVT1 15
EDMA channel 15

#define EDMA_CHA_TINT1L 16
EDMA channel 16

#define EDMA_CHA_TINT1H 17
EDMA channel 17

#define EDMA_CHA_18 18
EDMA channel 18

#define EDMA_CHA_19 19
EDMA channel 19

#define EDMA_CHA_20 20
EDMA channel 20

#define EDMA_CHA_21 21
EDMA channel 21

#define EDMA_CHA_22 22
EDMA channel 22

#define EDMA_CHA_23 23
EDMA channel 23

#define EDMA_CHA_24 24
EDMA channel 24

#define EDMA_CHA_25 25
EDMA channel 25

#define EDMA_CHA_26 26
EDMA channel 26

#define EDMA_CHA_27 27
EDMA channel 27

#define EDMA_CHA_VCPREVT0 28
EDMA channel 28

#define EDMA_CHA_VCPXEVT0 29
EDMA channel 29

#define EDMA_CHA_TCPREVT0 30
EDMA channel 30

#define EDMA_CHA_TCPXEVT0 31
EDMA channel 31

#define EDMA_CHA_UREVT 32
EDMA channel 32

#define EDMA_CHA_33 33
EDMA channel 33

#define EDMA_CHA_34 34
EDMA channel 34

#define EDMA_CHA_35 35
EDMA channel 35

#define EDMA_CHA_36 36
EDMA channel 36

#define EDMA_CHA_37 37
EDMA channel 37

#define EDMA_CHA_38 38
EDMA channel 38

#define EDMA_CHA_39 39
EDMA channel 39

#define EDMA_CHA_UXEVT 40
EDMA channel 40

#define EDMA_CHA_41 41
EDMA channel 41

#define EDMA_CHA_42 42
EDMA channel 42

#define EDMA_CHA_43 43
EDMA channel 43

#define EDMA_CHA_ICREVT 44
EDMA channel 44

#define EDMA_CHA_ICXEVT 45
EDMA channel 45

#define EDMA_CHA_46 46
EDMA channel 46

#define EDMA_CHA_47 47
EDMA channel 47

#define EDMA_CHA_GPINT0 48
EDMA channel 48

#define EDMA_CHA_GPINT1 49
EDMA channel 49

#define EDMA_CHA_GPINT2 50
EDMA channel 50

#define EDMA_CHA_GPINT3 51
EDMA channel 51

#define EDMA_CHA_GPINT4 52
EDMA channel 52

#define EDMA_CHA_GPINT5 53
EDMA channel 53

#define EDMA_CHA_GPINT6 54
EDMA channel 54

#define EDMA_CHA_GPINT7 55
EDMA channel 55

#define EDMA_CHA_GPINT8 56
EDMA channel 56

#define EDMA_CHA_GPINT9 57
EDMA channel 57

#define EDMA_CHA_GPINT10 58
EDMA channel 58

#define EDMA_CHA_GPINT11 59
EDMA channel 59

#define EDMA_CHA_GPINT12 60
EDMA channel 60

#define EDMA_CHA_GPINT13 61
EDMA channel 61

#define EDMA_CHA_GPINT14 62
EDMA channel 62

#define EDMA_CHA_GPINT15 63
EDMA channel 63

#define EDMA_CHA_ANY -1
Use this to open any EDMA channel

#define EDMA_CHA_CNT (_EDMA_CHA_CNT)
Number of EDMA channels

#define EDMA_HINV _EDMA_MK_HANDLE(0x00000000,0,0)
Invalid handle

#define EDMA_OPEN_ENABLE (0x00000002)
Enable flag passed to EDMA_open, enables the particular channel to service events

#define EDMA_OPEN_RESET (0x00000001)
Reset flag passed to EDMA_open API.

#define EDMA_TABLE_CNT (_EDMA_LINK_CNT)
Total number of PaRAM tables available

#define EDMA_TCC_SET   1
Macro for EDMA transfer completion code interrupt

#define NULL_FUNC   0
NULL function

#define EDMA_ATCC_SET   1
Macro for EDMA alternate transfer completion code interrupt

5.5 Typedefs

typedef Uint32 EDMA_Handle
EDMA handle returned by EDMA_open and EDMA_allocTable

Chapter 6
EMIFA Module

Topics

6.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within EMIFA module. A 64-bit external memory interface which is capable of interfacing to asynchronous peripherals (including SRAM, ROM, and Flash).

The EMIF module has a simple API for configuring the EMIF registers.

The EMIF provides a glue less interface to external memory devices including SDR and a wide variety of asynchronous devices.

The EMIF features supports following functionality:

- SDRAM controller
- ASync controller
- Little endian operation
- Full rate operation

6.2 Functions

This section lists the functions available in the EMIFA module.

6.2.1 CSL_emifaInit

CSL_Status CSL_emifaInit (CSL_EmifaContext * *pContext*)

Description
This is the initialization function for the EMIFA CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context.Context information for the
            instance. As EMIFA doesn't have any context based
            information user is expected to pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
This function should be called before using any of the CSL APIs.

Post Condition
The CSL for EMIFA is initialized.

Modifies
None

Example

```
 CSL_Status status;
...
status = CSL_emifaInit(NULL);
...
```

6.2.2 CSL_emifaOpen

CSL_EmifaHandle CSL_emifaOpen (CSL_EmifaObj * *pEmifaObj*, CSL_InstNum *emifaNum*, CSL_EmifaParam * *pEmifaParam*, CSL_Status * *pStatus*)

Description
This function returns the handle to the EMIFA instance. The open call sets up the data structures

for the particular instance of EMIFA. The handle returned by this call is input argument for rest of the EMIFA CSL APIs.

Arguments

`pEmifaObj`	`Pointer to the EMIFA instance object`
`emifaNum`	`Instance of the EMIFA to be opened.`
`pEmifaParam`	`Pointer to module specific parameters`
`pStatus`	`Pointer for returning status of the function call`

Return Value
`CSL_EmifaHandle`

- Valid EMIFA instance handle will be returned if status value is equal to CSL_SOK.

Pre Condition
CSL_emifaInit() must be called successfully before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- CSL_SOK - Valid EMIFA handle is returned.
- CSL_ESYS_FAIL - The EMIFA instance is invalid.
- `CSL_ESYS_INVPARAMS` – The object structure is invalid.

2. EMIFA object structure is populated.

Modifies
1. The status variable
2. EMIFA object structure

Example:

```
CSL_Status          status;
CSL_EmifaObj        emifaObj;
CSL_EmifaHandle     hEmifa;

CSL_emifaInit(NULL);

hEmifa = CSL_emifaOpen( &emifaObj, CSL_EMIFA, NULL, &status);
...
```

6.2.3 CSL_emifaClose

CSL_Status CSL_emifaClose (CSL_EmifaHandle *hEmifa*)

Description
This function closes the specified instance of EMIFA. This is a module level close required to invalidate the module handle. The module handle must not be used after this API call.

Arguments

```
hEmifa          Handle to the external memory interface instance
```

Return Value
`CSL_Status`

- `CSL_SOK` - external memory interface close successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Both *CSL_emifaInit()* and *CSL_emifaOpen()* must be called successfully in order before calling *CSL_emifaClose()*.

Post Condition
The external memory interface CSL APIs cannot be called until the external memory interface CSL is reopened again using CSL_emifaOpen().

Modifies
Obj structure values

Example

```
CSL_EmifaHandle   hEmifa;
CSL_Status        status;
//Initialize the Emifa CSL
...
//Open Emifa Module
...
status = CSL_emifaClose(hEmifa);
...
```

6.2.4 CSL_emifaHwSetupRaw

CSL_Status CSL_emifaHwSetupRaw **(** [CSL_EmifaHandle](#) ***hEmifa***,
[CSL_EmifaConfig](#) * ***config***
)

Description
This function initializes the device registers with the register-values provided through the Config data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hEmifa      Handle to the EMIFA external memory interface instance

config      Pointer to the config structure containing the
            device register values
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful

- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration structure pointer is not properly initialized

Pre Condition
Both CSL_*emifaInit* () and CSL_emifaOpen() must be called successfully in order before calling this function.

Post Condition
The registers of the specified EMIFA instance will be setup according to the values passed through the Config structure.

Modifies
Hardware registers of the EMIFA

Example

```
CSL_EmifaHandle    hEmifa;
CSL_EmifaConfig    config = CSL_EMIFA_CONFIG_DEFAULTS;
CSL_Status         status;
...
status = CSL_emifaHwSetupRaw(hEmifa, &config);
...
```

6.2.5 CSL_emifaHwSetup

CSL_Status CSL_emifaHwSetup (CSL_EmifaHandle *hEmifa*, CSL_EmifaHwSetup * *setup*)

Description
This function initializes the device registers with the appropriate values provided through the HwSetup data structure. For information passed through the HwSetup data structure, refer *CSL_EmifaHwSetup*.

Arguments

`hEmifa`	`Handle to the EMIFA external memory interface instance`
`setup`	`Pointer to setup structure which contains the information to program EMIFA to required state`

Return Value
`CSL_Status`

- `CSL_SOK` - Hwsetup of EMIFA is successful
- `CSL_ESYS_FAIL` - Invalid access type (asynchronous and synchronous)
- `CSL_ESYS_INVPARAMS` - Parameters are not valid
- `CSL_ESYS_BADHANDLE` - Handle is not valid

Pre Condition

Both *CSL_emifaInit()* and *CSL_emifaOpen()* must be called successfully in order before calling this function.

Post Condition
EMIFA registers are configured according to the hardware setup parameters.

Modifies
EMIFA registers

Example:

```
CSL_EmifaHandle    hEmifa;
CSL_EmifaAsync     asyncMem = CSL_EMIFA_ASYNCCFG_DEFAULTS;
CSL_EmifaAsyncWait asyncWait = CSL_EMIFA_ASYNCWAIT_DEFAULTS;
CSL_EmifaMemType   value;
CSL_EmifaHwSetup   hwSetup;
CSL_Status         status;

value.ssel        = 0;
value.async       = &asyncMem;
value.sync        = NULL;
hwSetup.asyncWait = &asyncWait;
hwSetup.ceCfg[0]  = &value;
hwSetup.ceCfg[1]  = NULL;
hwSetup.ceCfg[2]  = NULL;
hwSetup.ceCfg[3]  = NULL;

//Initialize and Open the Emifa CSL
...
//Open Emifa Module
status = CSL_emifaHwSetup(hEmifa, &hwSetup);
...
```

6.2.6 CSL_emifaGetHwSetup

CSL_Status CSL_emifaGetHwSetup	**(**	CSL_EmifaHandle	***hEmifa***,
		CSL_EmifaHwSetup *****	***setup***
	)		

Description
This function gets the current setup of the EMIFA. The status is returned through *CSL_EmifaHwSetup*. The obtaining of status is the reverse operation of *CSL_emifaHwSetup()* function.

Arguments

```
hEmifa          Handle to the EMIFA external memory interface
                instance

setup           Pointer to the hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware status call is successful
- `CSL_ESYS_FAIL` - Invalid access type (asynchronous and synchronous).
- `CSL_ESYS_INVPARAMS` - Parameters are not valid
- `CSL_ESYS_BADHANDLE` - Handle is not valid

Pre Condition
Both *CSL_emifaInit()* and *CSL_emifaOpen()* must be called successfully in order before calling *CSL_emifaGetHwSetup()*.

Post Condition
None

Modifies
Second parameter setup value

Example:
```
CSL_EmifaHandle    hEmifa;
CSL_Status         status;
CSL_EmifaHwSetup   hwSetup;
CSL_EmifaAsync     asyncMem;
CSL_EmifaMemType   value;
CSL_EmifaAsyncWait asyncWait;

value.ssel        = 0;
value.async       = &asyncMem;
value.sync        = NULL;
hwSetup.asyncWait = &asyncWait;
hwSetup.ceCfg[0]  = &value;
hwSetup.ceCfg[1]  = NULL;
hwSetup.ceCfg[2]  = NULL;
hwSetup.ceCfg[3]  = NULL;

//Initialize the Emifa CSL
...
//Open Emifa Module
...
status = CSL_emifaGetHwSetup(hEmifa, &hwSetup);
...
```

6.2.7 CSL_emifaHwControl

CSL_Status CSL_emifaHwControl	**(**	**CSL_EmifaHandle**	***hEmifa*,**
		CSL_EmifaHwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
Control operations for the EMIFA. For a particular control operation, the pointer to the corresponding data type needs to be passed as argument HwControl function call. All the arguments (structure elements included) passed to the HwControl function are inputs. For the list of commands supported and argument type that can be *void** casted and passed with a particular command refer to *CSL_EmifaHwControlCmd*.

Arguments

```
hEmifa          Handle to the EMIFA external memory interface
                instance
cmd             The command to this API indicates the action to be
                taken
arg             Optional argument as per the control command
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware control call is successful
- `CSL_ESYS_INVCMD` - Command is not valid
- `CSL_ESYS_BADHANDLE` - Handle is not valid

Pre Condition
Both *CSL_emifaInit()* and *CSL_emifaOpen()* must be called successfully in order before calling *CSL_emifaHwControl()* can be called. For the argument type that can be *void** casted and passed with a particular command refer to *CSL_EmifaHwControlCmd.*

Post Condition
EMIFA registers are configured according to the command passed.

Modifies
EMIFA registers

Example:

```
CSL_EmifaHandle  hEmifa;
CSL_Status       status;
Uint8            command = 0xE0;
...
status = CSL_emifaHwControl(hEmifa,
                            CSL_EMIFA_CMD_PRIO_RAISE,
                            (void*) &command);
...
```

6.2.8 CSL_emifaGetHwStatus

CSL_Status CSL_emifaGetHwStatus (CSL_EmifaHandle *hEmifa*, CSL_EmifaHwStatusQuery *query*, void * *response*)

Description
This function is used to read the current device configuration, status flags and the value present associated registers. User should allocate memory for the said data type and pass its pointer as an unadorned void* argument to the status query call. For details about the various status queries supported and the associated data structure to record the response, refer to *CSL_EmifaHwStatusQuery*.

Arguments

`hEmifa`	`Handle to the EMIFA external memory interface instance`
`query`	`The query to this API which indicates the status to be returned`
`response`	`Placeholder to return the status. void* casted`

Return Value
`CSL_Status`

- `CSL_SOK` - Successful on getting hardware status
- `CSL_ESYS_INVQUERY` - Query is not valid
- `CSL_ESYS_BADHANDLE` - Handle is not valid
- `CSL_ESYS_INVPARAMS` – The parameter passed is not valid

Pre Condition
Both *CSL_emifaInit()* and *CSL_emifaOpen()* must be called successfully in order before calling *CSL_emifaGetHwStatus()* can be called. For the argument type that can be *void** casted and passed with a particular command refer to *CSL_EmifaHwStatusQuery.*

Post Condition
None

Modifies
Third parameter response value

Example:

```
CSL_EmifaHandle   hEmifa;
CSL_Status        status;
Uint8             response;
...
status = CSL_emifaGetHwStatus(hEmifa,
                              CSL_EMIFA_QUERY_ENDIAN,
                              (void*) &response);
...
```

6.2.9 CSL_emifaGetBaseAddress

CSL_Status CSL_emifaGetBaseAddress	**(**	**CSL_InstNum**	***emifaNum*,**
		CSL_EmifaParam *	***pEmifaParam*,**
		CSL_EmifaBaseAddress *	***pBaseAddress***
	)		

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_emifaOpen() function call. This function is open for re-implementing if the user wants to modify the base

address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

`emifaNum`	`Specifies the instance of the EMIFA for which the base address is requested`
`pEmifaParam`	`Module specific parameters.`
`pBaseAddress`	`Pointer to the base address structure to return the base address details.`

Return Value
`CSL_Status`

- `CSL_SOK` - Successful, on getting the base address of EMIFA.
- `CSL_ESYS_FAIL` - The external memory interface instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
None.

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure

Example

```
CSL_Status            status;
CSL_EmifaBaseAddress  baseAddress;
...
status = CSL_emifaGetBaseAddress(CSL_EMIFA, NULL, &baseAddress);
...
```

6.3 Data Structures

This section lists the data structures available in the EMIFA module.

6.3.1 CSL_EmifaObj

Detailed Description
This Object contains the reference to the instance of EMIFA opened using the *CSL_emifaOpen()*. The pointer to this is passed to all EMIFA CSL APIs. CSL_emifaOpen() function initializes this structure based on the parameters passed.

Field Documentation

CSL_InstNum CSL_EmifaObj::perNum
This is the instance of EMIFA being referred to by this object

CSL_EmifaRegsOvly CSL_EmifaObj::regs
Pointer to the register overlay structure of the EMIFA

6.3.2 CSL_EmifaConfig

Detailed Description
EMIFA config structure, which is used in *CSL_emifaHwSetupRaw()* function. This is a structure of register values, rather than a structure of register field values like *CSL_EmifaHwSetup*.

Field Documentation

volatile Uint32 CSL_EmifaConfig::AWCC
Asynchronous Wait Cycle Configuration register

volatile Uint32 CSL_EmifaConfig::BPRIO
Burst Priority Register

volatile Uint32 CSL_EmifaConfig::CE2CFG
Chip Enable2 Configuration register

volatile Uint32 CSL_EmifaConfig::CE3CFG
Chip Enable3 Configuration register

volatile Uint32 CSL_EmifaConfig::CE4CFG
Chip Enable4 Configuration register

volatile Uint32 CSL_EmifaConfig::CE5CFG
Chip Enable5 Configuration register

volatile Uint32 CSL_EmifaConfig::INTMSK
Interrupt Masked Register

volatile Uint32 CSL_EmifaConfig::INTMSKCLR
Interrupt Mask Clear Register

volatile Uint32 CSL_EmifaConfig::INTMSKSET
Interrupt Mask Set Register

volatile Uint32 CSL_EmifaConfig::INTRAW
Interrupt Raw Register

6.3.3 CSL_EmifaContext

Detailed Description
EMIFA specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_EmifaContext::contextInfo
Context information of EMIFA external memory interface CSL passed as an argument to CSL_emifaInit(). Present implementation of EMIFA CSL doesn't have any context information; hence assigned NULL. The declaration is just a placeholder for future implementation.

6.3.4 CSL_EmifaHwSetup

Detailed Description
This has all the fields required to configure EMIFA at Power Up (after a Hardware Reset) or a Soft Reset. This structure is used to setup or obtain existing setup of EMIFA using *CSL_emifaHwSetup()* and*CSL_emifaGetHwSetup()* functions respectively.

Field Documentation

CSL_EmifaAsyncWait* CSL_EmifaHwSetup::asyncWait
Pointer to structure for configuring the Asynchronous Wait Cycle Configuration register

CSL_EmifaMemType* CSL_EmifaHwSetup::ceCfg[NUMCHIPENABLE]
Array of CSL_EmifaMemType* for configuring the Chip enable as Async or Sync memory type.

6.3.5 CSL_EmifaParam

Detailed Description
Module specific parameters. Present implementation of EMIFA CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_EmifaParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation. Passed as an argument to *CSL_emifaOpen()*.

6.3.6 CSL_EmifaBaseAddress

Detailed Description
This structure contains the base address information for the EMIFA instance.

Field Documentation

CSL_EmifaRegsOvly CSL_EmifaBaseAddress::regs
Base address of the configuration registers of the peripheral

6.3.7 CSL_EmifaAsync

Detailed Description
EMIFA Async structure. The pointer to this structure is a member to the structure CSL_EmifaMemType. CSL_EmifaAsync structure holds the value to be programmed into CE Configuration register when ssel=0 (i.e. asynchronous).

Field Documentation

Uint8 CSL_EmifaAsync::asize
Asynchronous Memory Size

Uint8 CSL_EmifaAsync::asyncRdyEn
Asynchronous Ready Input Enable

Uint8 CSL_EmifaAsync::rHold
Read Hold Width

Uint8 CSL_EmifaAsync::rSetup
Read Setup Width

Uint8 CSL_EmifaAsync::rStrobe
Read Strobe Width

Uint8 CSL_EmifaAsync::selectStrobe
Select Strobe Mode Enable

Uint8 CSL_EmifaAsync::weMode
Select WE Strobe Mode Enable

Uint8 CSL_EmifaAsync::wHold
Write Hold Width

Uint8 CSL_EmifaAsync::wSetup
Write Setup Width

Uint8 CSL_EmifaAsync::wStrobe
Write Strobe Width

6.3.8 CSL_EmifaSync

Detailed Description
EMIFA Sync structure. The pointer to this structure is a member to the structure CSL_EmifaMemType. CSL_EmifaSync structure holds the value to be programmed into CE Configuration register when ssel=1 (i.e. synchronous).

Field Documentation

Uint8 CSL_EmifaSync::chipEnExt
Synchronous Memory Chip Enable Extend

Uint8 CSL_EmifaSync::r_ltncy
Synchronous Memory Read Latency

Uint8 CSL_EmifaSync::readByteEn
Read Byte Enable enable

Uint8 CSL_EmifaSync::readEn
Synchronous Memory Read Enable Mode

Uint8 CSL_EmifaSync::sbsize
Synchronous Memory Device Size

Uint8 CSL_EmifaSync::w_ltncy
Synchronous Memory Write Latency

6.3.9 CSL_EmifaMemType

Detailed Description
EMIFA MemType structure. This structure defines the memory type of a particular chip enable. If a particular chip enable e.g. CE2 is to be configured as asynchronous memory, ssel must be 0, sync must be NULL and async must be a pointer to CSL_EmifaAsync structure with the proper values configured.

Field Documentation

CSL_EmifaAsync* CSL_EmifaMemType::async
Pointer to structure of asynchronous type. The pointer value should be NULL if the chip select value is synchronous.

Uint8 CSL_EmifaMemType::ssel
Synchronous/asynchronous memory select. Asynchronous memory mode when ssel is set to 0 and synchronous when ssel is 1.

CSL_EmifaSync* CSL_EmifaMemType::sync
Pointer to structure of synchronous type. The pointer value should be NULL if the chip select value is asynchronous.

6.3.10 CSL_EmifaAsyncWait

Detailed Description
EMIFA AsyncWait structure. This structure is a structure member of CSL_EmifaHwSetup. It holds the value to be programmed into Asynchronous Wait Cycle Configuration register. This is valid only for asynchronous (ssel = 0) memories.

Field Documentation

CSL_EmifaArdyPol CSL_EmifaAsyncWait::asyncRdyPol
Asynchronous Ready Pin Polarity

Uint8 CSL_EmifaAsyncWait::maxExtWait
Maximum Extended Wait cycles

Uint8 CSL_EmifaAsyncWait::turnArnd
Turn Around cycles

6.3.11 CSL_EmifaModIdRev

Detailed Description

EMIFA Module ID and Revision structure. This structure is used for querying the EMIFA module ID and revision.

Field Documentation

Uint8 CSL_EmifaModIdRev::majRev

EMIFA Major Revision

Uint8 CSL_EmifaModIdRev::minRev

EMIFA Minor Revision

Uint16 CSL_EmifaModIdRev::modId

EMIFA Module ID

6.4 Enumerations

This section lists the enumerations available in the EMIFA module.

6.4.1 CSL_EmifaArdyPol

enum CSL_EmifaArdyPol
Enumeration for bit field AP of Asynchronous Wait Cycle Configuration Register.

Enumeration values:

CSL_EMIFA_ARDYPOL_LOW	Strobe period extended when ARDY is low
CSL_EMIFA_ARDYPOL_HIGH	Strobe period extended when ARDY is high

6.4.2 CSL_EmifaHwStatusQuery

enum CSL_EmifaHwStatusQuery
Enumeration for queries passed to *CSL_emifaGetHwStatus()* This is used to get the status of different operations.

Enumeration values:

CSL_EMIFA_QUERY_REV_ID	Get the EMIFA module ID and revision numbers **Parameters:** *(*CSL_EmifaModIdRev *)
CSL_EMIFA_QUERY_ASYNC_TIMEOUT_EN	Get Asynchronous Timeout status i.e. enabled or not **Parameters:** *(*Uint8 *)
CSL_EMIFA_QUERY_ASYNC_TIMEOUT_STATUS	Get Asynchronous Timeout status in Interrupt Raw register **Parameters:** *(*Uint8 *)
CSL_EMIFA_QUERY_ENDIAN	Gets the EMIFA EMIF Endianness **Parameters:** *(*Uint8 *)

6.4.3 CSL_EmifaHwControlCmd

enum CSL_EmifaHwControlCmd
Enumeration for commands passed to *CSL_emifaHwControl()*.
This is used to select the commands to control the operations existing setup of EMIFA. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_EMIFA_CMD_ASYNC_TIMEOUT_CLEAR	Clears Asyn Timeout interrupt: no argument **Parameters:** *(* None)
CSL_EMIFA_CMD_ASYNC_TIMEOUT_DISABLE	Disables Asyn Timeout interrupt: no argument **Parameters:** *(* None)
CSL_EMIFA_CMD_ASYNC_TIMEOUT_ENABLE	Enables Asyn Timeout interrupt: no argument **Parameters:** *(* None)
CSL_EMIFA_CMD_PRIO_RAISE	Number of memory transfers after which the EMIFA momentarily raises the priority of old commands in the VBUSM Command FIFO **Parameters:** *(*Uint8 *)

6.4.4 CSL_EmifaMemoryType

enum CSL_EmifaMemoryType
Enumeration for bit field for memory type

Enumeration values:

CSL_EMIFA_MEMTYPE_ASYNC	Asynchronous memory type
CSL_EMIFA_MEMTYPE_SYNC	Synchronous memory type

6.5 Macros

**#define CSL_EMIFA_ASYNCCFG_DEFAULTS **
```
{\
    (Uint8)CSL_EMIFA_ASYNCCFG_SELECTSTROBE_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_WEMODE_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_ASYNCRDYEN_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_WSETUP_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_SSTROBE_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_WHOLD_DEFAULT,\
    (Uint8)CSL_EMIFA_ASYNCCFG_RSETUP_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_RSTROBE_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_RHOLD_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCCFG_ASIZE_DEFAULT \
}
```
The default values for EMIFA CEConfig for Async structure.

#define CSL_EMIFA_ASYNCCFG_SELECTSTROBE_DEFAULT 0x00
#define CSL_EMIFA_ASYNCCFG_WEMODE_DEFAULT 0x00
#define CSL_EMIFA_ASYNCCFG_ASYNCRDYEN_DEFAULT 0x00
#define CSL_EMIFA_ASYNCCFG_WSETUP_DEFAULT 0x0F
#define CSL_EMIFA_ASYNCCFG_SSTROBE_DEFAULT 0x3F
#define CSL_EMIFA_ASYNCCFG_WHOLD_DEFAULT 0x07
#define CSL_EMIFA_ASYNCCFG_RSETUP_DEFAULT 0x0F
#define CSL_EMIFA_ASYNCCFG_RSTROBE_DEFAULT 0x3F
#define CSL_EMIFA_ASYNCCFG_RHOLD_DEFAULT 0x07
#define CSL_EMIFA_ASYNCCFG_ASIZE_DEFAULT 0x00
The default value for EMIFA CEConfig for Async structure

**#define CSL_EMIFA_ASYNCWAIT_DEFAULTS **
```
{\
    (CSL_EmifaArdyPol)CSL_EMIFA_ARDYPOL_HIGH, \
    (Uint8)CSL_EMIFA_ASYNCWAIT_MAXEXTWAIT_DEFAULT, \
    (Uint8)CSL_EMIFA_ASYNCWAIT_TURNARND_DEFAULT \
}
```
The default values for EMIFA Async Wait structure.

#define CSL_EMIFA_ASYNCWAIT_MAXEXTWAIT_DEFAULT 0x80
#define CSL_EMIFA_ASYNCWAIT_TURNARND_DEFAULT 0x03
The default value for EMIFA Async Wait structure

**#define CSL_EMIFA_CONFIG_DEFAULTS **
```
{ \
    (Uint32)CSL_EMIFA_CE2CFG_SSEL0_RESETVAL, \
    (Uint32)CSL_EMIFA_CE3CFG_SSEL0_RESETVAL, \
    (Uint32)CSL_EMIFA_CE4CFG_SSEL0_RESETVAL, \
    (Uint32)CSL_EMIFA_CE5CFG_SSEL0_RESETVAL, \
    (Uint32)CSL_EMIFA_AWCC_RESETVAL, \
    (Uint32)CSL_EMIFA_INTRAW_RESETVAL, \
    (Uint32)CSL_EMIFA_INTMSK_RESETVAL, \
    (Uint32)CSL_EMIFA_INTMSKSET_RESETVAL, \
    (Uint32)CSL_EMIFA_INTMSKCLR_RESETVAL, \
    (Uint32)CSL_EMIFA_BPRIO_RESETVAL \
```

```
}
```

The default values for Config structure.

**#define CSL_EMIFA_SYNCCFG_DEFAULTS **

```
{\
    (Uint8)CSL_EMIFA_SYNCCFG_READBYTEEN_DEFAULT, \
    (Uint8)CSL_EMIFA_SYNCCFG_CHIPENEXT_DEFAULT, \
    (Uint8)CSL_EMIFA_SYNCCFG_READEN_DEFAULT, \
    (Uint8)CSL_EMIFA_SYNCCFG_WLTNCY_DEFAULT, \
    (Uint8)CSL_EMIFA_SYNCCFG_RLTNCY_DEFAULT, \
    (Uint8)CSL_EMIFA_SYNCCFG_SBSIZE_DEFAULT \
}
```

The default values for EMIFA CEConfig for Sync structure.

#define CSL_EMIFA_SYNCCFG_READBYTEEN_DEFAULT 0x00
#define CSL_EMIFA_SYNCCFG_CHIPENEXT_DEFAULT 0x00
#define CSL_EMIFA_SYNCCFG_READEN_DEFAULT 0x00
#define CSL_EMIFA_SYNCCFG_WLTNCY_DEFAULT 0x00
#define CSL_EMIFA_SYNCCFG_RLTNCY_DEFAULT 0x00
#define CSL_EMIFA_SYNCCFG_SBSIZE_DEFAULT 0x00

The default values for EMIFA CEConfig for Sync structure

#define NUMCHIPENABLE 0x4

Total number of chip enables for Async/Sync memories

6.6 Typedefs

typedef CSL_EmifaObj * CSL_EmifaHandle
This is a pointer to CSL_EmifaObj and is passed as the first parameter to all EMIFA CSL APIs.

Chapter 7
GPIO Module

Topics

7.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within GPIO module. General-purpose input/output port (GPIO) with programmable interrupt/event generation modes having 16-pins.

The GPIO peripheral provides 16 dedicated general-purpose pins that can be configured as either inputs or outputs. Each GPx pin configured as an input can directly trigger a CPU interrupt or a GPIO event. The properties and functionalities of the GPx pins are covered by a set of CSL APIs.

To use the GPIO pins, the user must first allocate a device using *CSL_gpioOpen(),* and then configure the Global Control register to determine the peripheral mode by using the configuration structure.

7.2 Functions

This section lists the functions available in the GPIO module.

7.2.1 CSL_gpioInit

CSL_Status CSL_gpioInit ([CSL_GpioContext](#) * *pContext*)

Description
This is the initialization function for the GPIO CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status *CSL_SOK*. It has been kept for future use.

Arguments

```
pContext   Pointer to module-context.Context information for the
           instance. As GPIO  doesn't have any context based
           information user is expected to pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
The CSL for GPIO is initialized

Modifies
None

Example

```
CSL_Status status;
...
status = CSL_gpioInit(NULL);
...
```

7.2.2 CSL_gpioOpen

[CSL_GpioHandle](#) CSL_gpioOpen ([CSL_GpioObj](#) * *pGpioObj*,
CSL_InstNum *gpioNum*,
[CSL_GpioParam](#) * *pGpioParam*,
CSL_Status * *pStatus*
)

Description
This function populates the peripheral data object for the GPIO instance and returns a handle to the instance. The open call sets up the data structures for the particular instance of GPIO device.

The device can be re-opened anytime after it has been normally closed if so required. The handle returned by this call is input as an essential argument for rest of the GPIO CSL APIs.

Arguments

`pGpioObj`	`Pointer to the GPIO instance object`
`gpioNum`	`Instance of the GPIO to which a handle is requested`
`pGpioParam`	`Pointer to module specific parameters`
`pStatus`	`Pointer for returning status of the function call`

Return Value

`CSL_GpioHandle`

- Valid GPIO instance handle will be returned if status value is equal to *CSL_SOK*.

Pre Condition
The GPIO must be successfully initialized via *CSL_gpioInit()* before calling this function

Post Condition
1. GPIO object structure is populated
2. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid gpio handle is returned
- `CSL_ESYS_FAIL` - The gpio instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Modifies
1. The status variable
2. GPIO object structure is populated

Example

```
CSL_Status          status;
CSL_GpioObj         gpioObj;
CSL_GpioHandle      hGpio;

//Initialize the gpio CSL
...
hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
...
```

7.2.3 CSL_gpioClose

CSL_Status CSL_gpioClose (CSL_GpioHandle *hGpio*)

Description
This function closes the specified instance of GPIO.

Arguments

```
hGpio            Handle to the GPIO instance
```

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before calling *CSL_gpioClose()*.

Post Condition
The GPIO CSL APIs can not be called until the GPIO CSL is reopened again using *CSL_gpioOpen()*.

Modifies
Obj structure values

Example

```
CSL_GpioHandle      hGpio;
CSL_Status          status;
CSL_GpioObj         gpioObj;
hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
...

status = CSL_gpioClose(hGpio);
...
```

7.2.4 CSL_gpioHwSetup

CSL_Status CSL_gpioHwSetup (CSL_GpioHandle ***hGpio***, CSL_GpioHwSetup * ***setup***)

Description
It configures the gpio registers as per the values passed in the hardware setup structure. This is a dummy API . Its is left for future implementation.

Arguments

```
hGpio          Handle to the GPIO instance

setup          Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns.

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before this function. The user has to allocate space for and fill in the main setup structure appropriately before calling this function.

Post Condition
GPIO registers are configured according to the hardware setup parameters.

Modifies
Registers of GPIO.

Example

```
CSL_GpioHandle    hGpio;
CSL_GpioObj       gpioObj;
CSL_GpioHwSetup   hwSetup;
CSL_Status        status;

hwSetup.extendSetup = NULL;
...
hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
status = CSL_gpioHwSetup(hGpio, &hwSetup);
...
```

7.2.5 CSL_gpioHwSetupRaw

CSL_Status CSL_gpioHwSetupRaw	**(**	CSL_GpioHandle	***hGpio*,**
		CSL_GpioConfig *****	***config***
	)		

Description
This function initializes the device registers with the register-values provided through the Config Data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hGpio        Handle to the Gpio instance

config       Pointer to config structure containing the
             device register values
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration is not properly initialized

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before calling this function.

Post Condition
The registers of the specified GPIO instance will be setup according to value passed.

Modifies
Hardware registers of the GPIO.

Example

```
CSL_GpioHandle      hGpio;
CSL_GpioConfig      config = CSL_GPIO_CONFIG_DEFAULTS;
CSL_Status          status;
...
status = CSL_gpioHwSetupRaw(hGpio, &config);
...
```

7.2.6 CSL_gpioGetHwSetup

CSL_Status CSL_gpioGetHwSetup **(** [CSL_GpioHandle] ***hGpio*****,**
[CSL_GpioHwSetup] ***** ***setup***
)

Description
This function gets the current setup of the GPIO. This is the reverse operation of *CSL_gpioHwSetup()* function. This is a dummy API . Its is left for future implementation.

Arguments

```
hGpio           Handle to the GPIO instance

setup           Pointer to setup structure which contains the
                setup information of GPIO.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns.

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before calling this function

Post Condition
None

Modifies
Second parameter setup value.

Example

```
CSL_GpioHandle       hGpio;
CSL_GpioHwSetup      setup;
CSL_Status           status;
```

```
...
status = CSL_gpioGetHwSetup(hGpio, &setup);
...
```

7.2.7 CSL_gpioHwControl

CSL_Status CSL_gpioHwControl (CSL_GpioHandle *hGpio*, CSL_GpioHwControlCmd *cmd*, void * *arg*)

Description
Control operations for the GPIO. For a particular control operation, the pointer to the corresponding data type needs to be passed as argument to HwControl function Call.

Arguments

hGpio	Handle to the GPIO instance
cmd	The command to this API indicates the action to be taken on GPIO.
arg	Optional argument as per the control command.

Return Value
CSL_Status

- CSL_SOK - Status info return successful.
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVCMD - Invalid command
- CSL_EGPIO_INVPARAM - Invalid pin number

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before calling this function

Post Condition
GPIO registers are configured according to the command passed

Modifies
The hardware registers of GPIO.

Example

```
CSL_GpioHandle     hGpio;
CSL_Status         status;
CSL_GpioObj        gpioObj;
//Initialize the gpio CSL
...
hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
...
status = CSL_gpioHwControl(hGpio, CSL_GPIO_CMD_BANK_INT_ENABLE,
```

```
                                        NULL);
    ...
```

7.2.8 CSL_gpioGetHwStatus

CSL_Status CSL_gpioGetHwStatus **(** [CSL_GpioHandle](#) ***hGpio***, [CSL_GpioHwStatusQuery](#) ***query***, **void *** ***response*** **)**

Description
This function is used to read the current device configuration, status flags and the value present associated registers. For details about the various status queries supported and the associated data structure to record the response, refer to CSL_GpioHwStatusQuery..

Arguments

`hGpio`	`Handle to the GPIO instance`
`query`	`The query to this API of GPIO which indicates the status to be returned.`
`response`	`Placeholder to return the status.`

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware status call is successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY`- Invalid Query
- `CSL_ESYS_INVPARAMS` – Invalid Parameters

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before calling this function

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_GpioHandle          hGpio;
Uint32                  response;
CSL_Status              status;
CSL_GpioObj             gpioObj;

//Initialize the gpio CSL
...
hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
```

```
status = CSL_gpioGetHwStatus(hGpio, CSL_GPIO_QUERY_BINTEN_STAT,
                             &response);
...
```

7.2.9 CSL_gpioGetBaseAddress

CSL_Status CSL_gpioGetBaseAddress	**(**	**CSL_InstNum**	***gpioNum*****,**
		CSL_GpioParam *	***pGpioParam*****,**
		CSL_GpioBaseAddress *	***pBaseAddress***
	)		

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_gpioOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

gpioNum	Specifies the instance of GPIO to be opened.
pGpioParam	Module specific parameters.
pBaseAddress	Pointer to baseaddress structure containing base address details.

Return Value
CSL_Status

- CSL_SOK - Successfull on getting the base address of GPIO.
- CSL_ESYS_FAIL -The instance number is invalid.
- CSL_ESYS_INVPARAMS - Invalid Parameter

Pre Condition
None

Post Condition
Base Address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status            status;
CSL_GpioBaseAddress   baseAddress;
...
status = CSL_gpioGetBaseAddress(CSL_GPIO, NULL, &baseAddress);
```

7.3 Data Structures

This section lists the data structures available in the GPIO module.

7.3.1 CSL_GpioObj

Detailed Description

This object contains the reference to the instance of GPIO opened using the *CSL_gpioOpen()*. The pointer to this is passed to all GPIO CSL APIs. This structure has the fields required to configure GPIO. It should be initialized as per requirements of and passed on to the setup function

Field Documentation

CSL_InstNum CSL_GpioObj::gpioNum
This is the instance of GPIO being referred to by this object

CSL_GpioRegsOvly CSL_GpioObj::regs
Pointer to the register overlay structure of the GPIO

Uint8 CSL_GpioObj::numPins
This is the maximum number of pins supported by this instance of GPIO

7.3.2 CSL_GpioConfig

Detailed Description

Config structure of GPIO. This is used to configure GPIO using *CSL_gpioHwSetupRaw()* function. This is a structure of register values, rather than a structure of register field values like CSL_GpioHwSetup.

Field Documentation

volatile Uint32 CSL_GpioConfig::BINTEN
GPIO Interrupt Per-Bank Enable Register

volatile Uint32 CSL_GpioConfig::CLR_DATA
GPIO Clear Data Register

volatile Uint32 CSL_GpioConfig::CLR_FAL_TRIG
GPIO Clear Falling Edge Interrupt Register

volatile Uint32 CSL_GpioConfig::CLR_RIS_TRIG
GPIO Clear Rising Edge Interrupt Register

volatile Uint32 CSL_GpioConfig::DIR
GPIO Direction Register

volatile Uint32 CSL_GpioConfig::OUT_DATA
GPIO Output Data Register

volatile Uint32 CSL_GpioConfig::SET_DATA
GPIO Set Data Register

volatile Uint32 CSL_GpioConfig::SET_FAL_TRIG
GPIO Set Falling Edge Interrupt Register

volatile Uint32 CSL_GpioConfig::SET_RIS_TRIG
GPIO Set Rising Edge Interrupt Register

7.3.3 CSL_GpioContext

Detailed Description
GPIO specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_GpioContext::contextInfo
Context information of GPIO CSL passed as an argument to *CSL_gpioInit()*. Present implementation of GPIO CSL doesn't have any context information; hence assigned NULL. The declaration is just a placeholder for future implementation.

7.3.4 CSL_GpioParam

Detailed Description
GPIO specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_GpioParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

7.3.5 CSL_GpioHwSetup

Detailed Description
Input parameters for setting up GPIO during startup. This is just a placeholder as GPIO is a simple module, which doesn't require any setup

Field Documentation

void* CSL_GpioHwSetup::extendSetup
The extendSetup is just a placeholder for future implementation.

7.3.6 CSL_GpioBaseAddress

Detailed Description
Base-address of the Configuration registers of GPIO.

Field Documentation

CSL_GpioRegsOvly CSL_GpioBaseAddress::regs

Base address of the configuration registers of the peripheral

7.3.7 CSL_GpioPinConfig

Detailed Description
Input parameters for configuring a GPIO pin. This is used to configure the direction and edge detection.

Field Documentation

CSL_GpioDirection CSL_GpioPinConfig::direction
Direction for GPIO pin

CSL_GpioPinNum CSL_GpioPinConfig::pinNum
GPIO Pin Number

CSL_GpioTriggerType CSL_GpioPinConfig::trigger
GPIO pin edge detection

7.3.8 CSL_GpioPinData

Detailed Description
This is used for getting a specific pin status.

Field Documentation

CSL_GpioPinNum CSL_GpioPinData::pinNum
Pin number for GPIO bank

Int16 CSL_GpioPinData::pinVal
Pin value of the specified pin number

7.4 Enumerations

7.4.1 CSL_GpioDirection

enum CSL_GpioDirection
Enumeration for configuring GPIO pin direction.

Enumeration values:

CSL_GPIO_DIR_OUTPUT	Output pin
CSL_GPIO_DIR_INPUT	Input pin

7.4.2 CSL_GpioTriggerType

enum CSL_GpioTriggerType
Enumeration for configuring GPIO pin edge detection.

Enumeration values:

CSL_GPIO_TRIG_CLEAR_EDGE	No edge detection
CSL_GPIO_TRIG_RISING_EDGE	Rising edge detection
CSL_GPIO_TRIG_FALLING_EDGE	Falling edge detection
CSL_GPIO_TRIG_DUAL_EDGE	Dual edge detection

7.4.3 CSL_GpioHwControlCmd

enum CSL_GpioHwControlCmd
Enumeration for control commands passed to *CSL_gpioHwControl()*.
This is the set of commands that are passed to the *CSL_gpioHwControl()* with an optional argument type-casted to *void** . The arguments to be passed with each enumeration (if any) are specified next to the enumeration.

Enumeration values:

CSL_GPIO_CMD_BANK_INT_ENABLE	Enables interrupt on bank. **Parameters:** (None)
CSL_GPIO_CMD_BANK_INT_DISABLE	Disables interrupt on bank. **Parameters:** (None)
CSL_GPIO_CMD_CONFIG_BIT	Configures GPIO pin direction and edge detection properties. **Parameters:** (CSL_GpioPinConfig*)
CSL_GPIO_CMD_SET_BIT	Changes output state of GPIO pin to logic-1. **Parameters:** (CSL_GpioPinNum*)

CSL_GPIO_CMD_CLEAR_BIT	Changes output state of GPIO pin to logic-0. **Parameters:** (CSL_GpioPinNum*)
CSL_GPIO_CMD_GET_INPUTBIT	Gets the state of input pins on bank The "data" field acts as output parameter reporting the input state of the GPIO pins on the bank. **Parameters:** (CSL_BitMask16*)
CSL_GPIO_CMD_GET_OUTDRVSTATE	Gets the state of output pins on bank. The "data" field acts as output parameter reporting the output drive state of the GPIO pins on the bank. **Parameters:** (CSL_BitMask16*)
CSL_GPIO_CMD_GET_BIT	Gets the state of input pin on bank. **Parameters:** (CSL_GpioPinData*)
CSL_GPIO_CMD_ENABLE_DISABLE_OUTBIT	Changes output state of GPIO pin to logic-1 and logic-0 according to the parameter passed. **Parameters:** (CSL_GpioPinData*)

7.4.4 CSL_GpioHwStatusQuery

enum CSL_GpioHwStatusQuery
Enumeration for queries passed to *CSL_GpioGetHwStatus()*.
This is used to get the status of different operations. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_GPIO_QUERY_BINTEN_STAT	Queries GPIO bank interrupt enable status. **Parameters:** (CSL_BitMask16*)

7.4.5 CSL_GpioPinNum

enum CSL_GpioPinNum
Enumeration used to specify the GPIO pin numbers

Enumeration values:

CSL_GPIO_PIN0	Gpio pin 0
CSL_GPIO_PIN1	Gpio pin 1
CSL_GPIO_PIN2	Gpio pin 2
CSL_GPIO_PIN3	Gpio pin 3
CSL_GPIO_PIN4	Gpio pin 4
CSL_GPIO_PIN5	Gpio pin 5

CSL_GPIO_PIN6	Gpio pin 6
CSL_GPIO_PIN7	Gpio pin 7
CSL_GPIO_PIN8	Gpio pin 8
CSL_GPIO_PIN9	Gpio pin 9
CSL_GPIO_PIN10	Gpio pin 10
CSL_GPIO_PIN11	Gpio pin 11
CSL_GPIO_PIN12	Gpio pin 12
CSL_GPIO_PIN13	Gpio pin 13
CSL_GPIO_PIN14	Gpio pin 14
CSL_GPIO_PIN15	Gpio pin 15

7.5 Macros

#define CSL_EGPIO_INVPARAM CSL_EGPIO_FIRST
Value for invalid argument

**#define CSL_GPIO_CONFIG_DEFAULTS **
```
{       \
    CSL_GPIO_BINTEN_RESETVAL ,           \
    CSL_GPIO_DIR_RESETVAL,               \
    CSL_GPIO_OUT_DATA_RESETVAL,          \
    CSL_GPIO_SET_DATA_RESETVAL,          \
    CSL_GPIO_CLR_DATA_RESETVAL,          \
    CSL_GPIO_SET_RIS_TRIG_RESETVAL,      \
    CSL_GPIO_CLR_RIS_TRIG_RESETVAL,      \
    CSL_GPIO_SET_FAL_TRIG_RESETVAL,      \
    CSL_GPIO_CLR_FAL_TRIG_RESETVAL,      \
}
```
Default values for GPIO Config structure.

7.6 Typedefs

typedef CSL_GpioObj * CSL_GpioHandle
This is a pointer to CSL_GpioObj and is passed as the first parameter to all GPIO CSL APIs

Chapter 8
HPI Module

Topics

8.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within HPI module. Host Port Interface supports 16-bit or 32-bit which is user configurable.

Host Port Interface (HPI) provides a parallel port through which an external host processor can access a CPU’s memory space. The HPI enables a host device and CPU to exchange information via internal or external memory. Connectivity to the CPU’s memory space is provided through the HPI’s Vbus master interface. The Vbus master initiates CPU memory accesses through the EDMA. Dedicated address and Data registers (HPIA and HPID) within the HPI provide the data path between the external host interface and the Vbus master interface. A HPI control register (HPIC) is available to the host and the CPU for various configuration and interrupt functions.

The HPI module has a simple API for configuring the HPI registers. Functions are provided for reading HPI status bits and setting interrupt events. In this write and Read memory addresses can be accessed. A parallel interface that the CPU uses to communicate with a host processor.

HPI is an API module used for configuring the HPI registers. Functions are provided for reading HPI status bits and setting interrupt events.

8.2 Functions

This section lists the functions available in the HPI module.

8.2.1 CSL_hpiInit

CSL_Status CSL_hpiInit (CSL_HpiContext* *pContext*)

Description
This is the initialization function for the HPI CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context. As HPI doesn't have
            any context based information, user is expected to
            pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
The CSL for HPI is initialized.

Modifies
None

Example

```
CSL_Status          status;
...
status = CSL_hpiInit(NULL);
...
```

8.2.2 CSL_hpiOpen

CSL_HpiHandle CSL_hpiOpen (CSL_HpiObj * *pHpiObj*, CSL_InstNum *hpiNum*, CSL_HpiParam * *pHpiParam*, CSL_Status * *pStatus*)

Description
This function returns the handle to the HPI controller instance. This handle is passed to all other CSL APIs.

Arguments

`pHpiObj`	`Pointer to the object that holds reference to the instance of HPI requested after the call.`
`hpiNum`	`Instance of HPI to which a handle is requested. There is only one instance of the HPI available. So, the value for this parameter will be CSL_HPI always.`
`pHpiParam`	`Pointer to module specific parameters.`
`pStatus`	`Status of the function call`

Return Value
`CSL_HpiHandle`

- Valid HPI handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The HPI must be successfully initialized via CSL_*hpiInit*() before calling this function

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid HPI handle is returned
- `CSL_ESYS_FAIL` - The HPI instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

2. HPI object structure is populated.

Modifies
1. The status variable
2. HPI object structure

Example

```
CSL_Status          status;
CSL_HpiObj          hpiObj;
CSL_HpiHandle       hHpi;
...

hHpi = CSL_hpiOpen(&hpiObj, CSL_HPI, NULL, &status);
...
```

8.2.3 CSL_hpiClose

CSL_Status CSL_hpiClose ([CSL_HpiHandle] *hHpi*)

Description
This function closes the specified instance of HPI.

Arguments

hHpi	Handle to the HPI

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling CSL_hpiClose().

Post Condition
The HPI CSL APIs can not be called until the HPI CSL is reopened again using CSL_hpiOpen().

Modifies
Obj structure values

Example

```
CSL_HpiHandle       hHpi;
CSL_Status          status;
...
status = CSL_hpiClose(hHpi);
...
```

8.2.4 CSL_hpiHwSetup

CSL_Status CSL_hpiHwSetup **(** [CSL_HpiHandle] ***hHpi*****,**
CSL_HpiHwSetup * ***hwSetup***
)

Description
It configures the HPI registers as per the values passed in the hardware setup structure.

Arguments

hHpi	Handle to the HPI
hwSetup	Pointer to hardware setup structure

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
Both CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling this function.

Post Condition
HPI registers are configured according to the hardware setup parameters.

Modifies
HPI registers

Example

```
CSL_Status          status;
CSL_HpiHwSetup      hwSetup;
CSL_HpiHandle       hHpi;
hwSetup.hpiCtrl = (CSL_HpiCtrl)0x80;
...

status = CSL_hpiHwSetup(hHpi, &hwSetup);
...
```

8.2.5 CSL_hpiHwControl

CSL_Status CSL_hpiHwControl **(** [CSL_HpiHandle] ***hHpi*,**
[CSL_HpiHwControlCmd] ***cmd*,**
void * ***arg***
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of HPI.

Arguments

`hHpi`	`Handle to the HPI instance`
`cmd`	`The command to this API indicates the action to be taken on HPI.`
`arg`	`An optional argument.`

Return Value
`CSL_Status`

- `CSL_SOK` - Command successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVCMD` - Invalid command
- CSL_ESYS_INVPARAMS – Invalid parameters

Pre Condition
CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling CSL_hpiHwControl().

Post Condition
HPI registers are configured according to the command passed.

Modifies
The hardware registers of HPI.

Example

```
CSL_HpiHandle        hHpi;
CSL_Status           status;
CSL_HpiHwControlCmd  cmd = CSL_HPI_CMD_SET_HINT;

...

status = CSL_hpiHwControl(hHpi, cmd, NULL);
...
```

8.2.6 CSL_hpiGetHwStatus

CSL_Status CSL_hpiGetHwStatus	(	CSL_HpiHandle	***hHpi***,
		CSL_HpiHwStatusQuery	***query***,
		void *	***response***
	)		

Description
Gets the status of the different operations of HPI.

Arguments

```
hHpi          Handle to the HPI instance
query         The query to this API of HPI which indicates the
              status to be returned.
response      Placeholder to return the status.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Query successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - The Query passed is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling CSL_hpiGetHwStatus().

Post Condition
None

Modifies
Third parameter response value

Example

```
CSL_HpiHandle          hHpi;
CSL_HpiHwStatusQuery   query = CSL_HPI_QUERY_HRDY;
Uint32                 response;
CSL_Status             status;
...

status = CSL_hpiGetHwStatus(hHpi, query, &response);
...
```

8.2.7 CSL_hpiHwSetupRaw

CSL_Status CSL_hpiHwSetupRaw **(** CSL_HpiHandle ***hHpi*****,**
CSL_HpiConfig * ***config***
)

Description
This function initializes the device registers with the register-values provided through the Config data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hHpi        Handle to the HPI instance

config      Pointer to Config structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration is not properly initialized

Pre Condition

CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling CSL_hpiHwSetupRaw().

Post Condition
The registers of the specified HPI instance will be setup according to input configuration structure values.

Modifies
Hardware registers of the specified HPI instance.

Example

```
CSL_HpiHandle       hHpi;
CSL_HpiConfig       config = CSL_HPI_CONFIG_DEFAULTS;
```

```
CSL_Status          status;
...

status = CSL_hpiHwSetupRaw(hHpi, &config);
...
```

8.2.8 CSL_hpiGetHwSetup

CSL_Status CSL_hpiGetHwSetup **(** **CSL_HpiHandle** ***hHpi*****,**
CSL_HpiHwSetup * ***hwSetup***
)

Description
It retrieves the hardware setup parameters of the HPI specified by the given handle.

Arguments

```
hHpi          Handle to the hpi

hwSetup       Pointer to the hardware setup structure
```

Return Value CSL_Status

- CSL_SOK - Retrieving the hardware setup parameters is successful
- CSL_ESYS_BADHANDLE - The handle is passed is invalid
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
CSL_hpiInit() and CSL_hpiOpen() must be called successfully in order before calling CSL_hpiGetHwSetup().

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
hwSetup variable

Example

```
CSL_HpiHandle   hHpi;
CSL_HpiHwSetup  hwSetup;
CSL_Status     status;
...

status = CSL_hpiGetHwSetup(hHpi, &hwSetup);
...
```

8.2.9 CSL_hpiGetBaseAddress

CSL_Status CSL_hpiGetBaseAddress (CSL_InstNum ***hpiNum*****,**
CSL_HpiParam * ***pHpiParam*****,**
CSL_HpiBaseAddress * ***pBaseAddress***
)

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_hpiOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral. MMRs go to an alternate location.

Arguments

hpiNum	Specifies the instance of the hpi to be opened.
pHpiParam	Pointer to module specific parameters.
pBaseAddress	Pointer to base address structure containing base address details.

Return Value
CSL_Status

- CSL_SOK - Successful, on getting the base address of HPI.
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status           status;
CSL_HpiBaseAddress   baseAddress;
...

status = CSL_hpiGetBaseAddress(CSL_HPI, NULL, &baseAddress);
...
```

8.3 Data Structures

This section lists the data structures available in the HPI module.

8.3.1 CSL_HpiObj

Detailed Description

This structure/object holds the context of the instance of HPI opened using CSL_hpiOpen() function. Pointer to this object is passed as HPI Handle to all HPI CSL APIs. CSL_hpiOpen() function initializes this structure based on the parameters passed.

Field Documentation

CSL_HpiRegsOvly CSL_HpiObj::regs
Pointer to the register overlay structure of the HPI

CSL_InstNum CSL_HpiObj::hpiNum
Instance of HPI being referred by this object

8.3.2 CSL_HpiConfig

Detailed Description

Config-structure used to configure the HPI using CSL_hpiHwSetupRaw(). This is a structure of register values, rather than a structure of register field values like CSL_HpiHwSetup.

volatile Uint32 CSL_HpiConfig::PWREMU_MGMT
Power and Emulation Management Register

volatile Uint32 CSL_HpiConfig:: HPIC
Host Port Interface Control Register

volatile Uint32 CSL_HpiConfig:: HPIAW
Host Port Interface Write Address Register

volatile Uint32 CSL_HpiConfig:: HPIAR
Host Port Interface Read Address Register

8.3.3 CSL_HpiContext

Detailed Description

HPI specific context information. Present implementation of HPI CSL doesn't have any context information.

Field Documentation

Uint32 CSL_HpiContext::contextInfo
Context information of HPI CSL. The declaration is just a placeholder for future implementation.

8.3.4 CSL_HpiParam

Detailed Description

HPI specific parameters. Present implementation of HPI CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask32 CSL_HpiParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

8.3.5 CSL_HpiHwSetup

Detailed Description
The structure contains the HPI hardware setup.

Field Documentation

Uint32 CSL_HpiHwSetup::emu
Emulation Mode parameter

CSL_HpiAddrCfg CSL_HpiHwSetup::hpiAddr
Host port Interface Read & Write Address Register

CSL_HpiCtrl CSL_HpiHwSetup::hpiCtrl
Host port Interface control Register

8.3.6 CSL_HpiBaseAddress

Detailed Description
This structure contains the base-address information for the peripheral instance of the HPI.

Field Documentation

CSL_HpiRegsOvly CSL_HpiBaseAddress::regs
Base-address of the configuration registers of the HPI peripheral

8.3.7 CSL_HpiAddrCfg

Detailed Description
Structure configures Host Port Interface Write and Read Address.

Field Documentation

Uint32 CSL_HpiAddrCfg::hpiaReadAddr
Host Port Interface Read Address

Uint32 CSL_HpiAddrCfg::hpiaWrtAddr
Host Port Interface Write Address

8.4 Enumerations

This section lists the enumerations available in the HPI module.

8.4.1 CSL_HpiHwStatusQuery

enum CSL_HpiHwStatusQuery
Enumeration for hardware status query commands

Enumeration values:

CSL_HPI_QUERY_HRDY	Query the current value of Host Ready. **Parameters:** *(Uint32* *)
CSL_HPI_QUERY_FETCH	Query the current value of HPI Fetch. **Parameters:** *(Uint32* *)
CSL_HPI_QUERY_HPI_RST	Query the current value of HPI Reset. **Parameters:** *(Uint32* *)
CSL_HPI_QUERY_HWOB_STAT	Query the current value of Half-word ordering status. **Parameters:** *(Uint32* *)

8.4.2 CSL_HpiHwControlCmd

enum CSL_HpiHwControlCmd

Enumeration values:

CSL_HPI_CMD_SET_DSP_INT	Sets the HPIC Host-to-DSP Interrupt. **Parameters:** *(None)*
CSL_HPI_CMD_RESET_DSP_INT	Reset the HPIC Host-to-DSP Interrupt. **Parameters:** *(None)*
CSL_HPI_CMD_SET_HINT	Sets the HPIC DSP-to-Host Interrupt. **Parameters:** *(None)*
CSL_HPI_CMD_RESET_HINT	Reset the HPIC DSP-to-Host Interrupt. **Parameters:** *(None)*

8.4.3 CSL_HpiCtrl

enum CSL_HpiCtrl
The control commands of HPI.

Enumeration values:

CSL_HPI_HWOB	Half-word Ordering Bit
CSL_HPI_DSP_INT	Host-to-DSP Interrupt
CSL_HPI_HINT	DSP-to-Host Interrupt
CSL_HPI_HRDY	Host Ready
CSL_HPI_FETCH	Host Fetch
CSL_HPI_RESET	CPU Core Reset
CSL_HPI_HPI_RST	HPI Reset
CSL_HPI_HWOB_STAT	Half-word ordering bit status
CSL_HPI_DUAL_HPIA	Dual HPIA mode configuration bit
CSL_HPI_HPIA_RW_SEL	HPIA register select bit

8.5 Macros

**#define CSL_HPI_CONFIG_DEFAULTS **

```
{ \
    CSL_HPI_PWREMU_MGMT_RESETVAL, \
    CSL_HPI_HPIC_RESETVAL, \
    CSL_HPI_HPIAW_RESETVAL, \
    CSL_HPI_HPIAR_RESETVAL \
}
```

Default values for Config structure

8.6 Typedefs

typedef CSL_HpiObj * CSL_HpiHandle
This data type is used to return the handle to the CSL of the HPI.

Chapter 9
I2C Module

Topics

9.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within I2C module. The I2C ports allows the DSP to easily control peripheral devices and communicate with a host processor.

The inter-integrated circuit (I2C) module provides an interface between a DSP and other devices of Inter−IC bus (I2C−bus).

9.2 Functions

This section lists the functions available in the I2C module.

9.2.1 CSL_i2cInit

CSL_Status CSL_i2cInit (CSL_I2cContext * *pContext*)

Description
This is the initialization function for the I2C CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Context information for the instance.  As I2C
            doesn't have any context based information user is
            expected to pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
The CSL for I2C is initialized

Modifies
None

Example

```
CSL_Status        status;
...
status = CSL_i2cInit(NULL);
...
```

9.2.2 CSL_i2cOpen

CSL_I2cHandle CSL_i2cOpen (CSL_I2cObj * *pI2cObj*, CSL_InstNum *i2cNum*, CSL_I2cParam * *pI2cParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the instance and returns a handle to the

instance. The open call sets up the data structures for the particular instance of I2C device. The device can be re-opened anytime after it has been normally closed if so required. The handle returned by this call is input argument for rest of the I2C CSL APIs.

Arguments

`pI2cObj`	`Pointer to the I2C instance object`
`i2cNum`	`Instance of the I2C to be opened.`
`pI2cParam`	`Pointer to module specific parameters`
`pStatus`	`Pointer for returning status of the function call`

Return Value
`CSL_I2cHandle`

- Valid I2C handle will be returned if status value is equal to CSL_SOK.

Pre Condition
CSL_i2cInit() must be called successfully.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid I2C instance handle will be returned.
- `CSL_ESYS_INVPARAMS` – Invalid parameter.
- `CSL_ESYS_FAIL – The I2C` instance is invalid.

2. I2C object structure is populated.

Modifies
1. The status variable
2. I2C object structure

Example:
```
CSL_Status        status;
CSL_I2cObj        i2cObj;
CSL_I2cHandle     hI2c;
...
hI2c = CSL_i2cOpen(&i2cObj,CSL_I2C,NULL,&status);
...
```

9.2.3 CSL_i2cClose

CSL_Status CSL_i2cClose (CSL_I2cHandle *hI2c*)

Description
This function closes the specified instance of I2C.

Arguments

`hI2c`	`Handle to the I2C`

Return Value
`CSL_Status`

- `CSL_SOK` - Close Successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling *CSL_i2cClose()*.

Post Condition
The I2C CSL APIs can not be called until the I2C CSL is reopened again using *CSL_i2cOpen()*.

Modifies
Obj structure values

Example

```
CSL_I2cHandle   hI2c;
CSL_Status      status;
...
status = CSL_i2cClose(hI2c);
...
```

9.2.4 CSL_i2cHwSetup

CSL_Status CSL_i2cHwSetup **(** CSL_I2cHandle ***hI2c*,** CSL_I2cHwSetup ***** ***setup*** **)**

Description
This function initializes the device registers with the appropriate values provided through the HwSetup Data structure. After the Setup is completed, the device is ready for operation. For information passed through the HwSetup Data structure, refer *CSL_I2cHwSetup*.

Arguments

`hI2c`	`Handle to the I2C`
`setup`	`Pointer to the setup structure which contains the setup information of I2C`

Return Value
`CSL_Status`

- `CSL_SOK` - HwSetup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function. The user has to allocate space for and fill in the main setup structure appropriately before calling this function.

Post Condition
I2C registers are configured according to the hardware setup parameters.

Modifies
I2C registers will be setup according to value passed.

Example

```
CSL_I2cHandle  hI2c;
CSL_I2cHwSetup hwSetup;
CSL_Status     status;

...
hwSetup.mode       = CSL_I2C_MODE_MASTER;
hwSetup.dir        = CSL_I2C_DIR_TRANSMIT;
hwSetup.addrMode   = CSL_I2C_ADDRSZ_SEVEN;
hwSetup.sttbyteen  = CSL_I2C_STB_DISABLE;

status = CSL_i2cHwSetup(hI2c, &hwSetup);
...
```

9.2.5 CSL_i2cGetHwSetup

CSL_Status CSL_i2cGetHwSetup (CSL_I2cHandle *hI2c*,
CSL_I2cHwSetup * *setup*
)

Description
This function gets the current setup of the I2C. The status is returned through *CSL_I2cHwSetup*. The operation of obtaining the status is reverse operation of *CSL_I2cHwSetup()* function.

Arguments

hI2c	Handle to the I2C
setup	Pointer to the hardware setup structure

Return Value
CSL_Status

- CSL_SOK - Retrieving the hardware setup parameters is successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function.

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
Second Parameter setup value

Example

```
CSL_Status status;
CSL_I2cHandle   hI2c;
CSL_I2cHwSetup  hwSetup;
...
status = CSL_i2cGetHwSetup(hI2c, &hwSetup);
...
```

9.2.6 CSL_i2cHwControl

CSL_Status CSL_i2cHwControl	**(**	CSL_I2cHandle	***hI2c*,**
		CSL_I2cHwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
Control operations for the I2C. For a particular control operation, the pointer to the corresponding data type need to be passed as argument to HwControl function call. For the list of commands supported and argument type that can be *void** casted and passed with a particular command refer to *CSL_I2cHwControlCmd*.

Arguments

```
hI2c        Handle to the I2C instance

cmd         The command to this API indicates the action to be
            taken on I2C

arg         An optional argument
```

Return Value
`CSL_Status`

- `CSL_SOK` - Command successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function.

Post Condition
I2C registers are configured according to the command passed

Modifies
The hardware registers of I2C.

Example

```
CSL_I2cHandle         hI2c;
CSL_I2cHwControlCmd   cmd = CSL_I2C_CMD_SET_SLAVE_ADDR;
Uint16                arg = 0x3FF;
CSL_Status            status;
...
status = CSL_i2cHwControl(hI2c, cmd, &arg);
...
```

9.2.7 CSL_i2cRead

CSL_Status CSL_i2cRead	**(**	CSL_I2cHandle	***hI2c*,**
		void *	***buf***
	)		

Description
This function reads I2C data.

Arguments

```
hI2c            Handle to I2C instance

buf             Buffer to store the data read
```

Return Value
`CSL_Status`

- `CSL_SOK` – Read operation Successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function.

Post Condition
None

Modifies
None

Example:

```
Uint8         outData;
CSL_Status    status;
CSL_I2cHandle hI2c;
...
/* Define I2C object and HwSetup structure and
   initialize the same */
...

/* Init, Open, HwSetup successfully done in that order */
...
```

```
status = CSL_i2cRead(hI2c, &outData);
...
```

9.2.8 CSL_i2cWrite

CSL_Status CSL_i2cWrite	**(**	**CSL_I2cHandle**	***hI2c*,**
		void *	***buf***
	)		

Description
This function writes the specified data into I2C data register.

Arguments

```
hI2c            Handle to I2C instance

buf             Data to be written
```

Return Value
`CSL_Status`

- `CSL_SOK` – Write success (does not verify written data)
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function.

Post Condition
Data is written to I2C data register

Modifies
I2C register

Example:

```
Uint8         inData;
CSL_Status    status;
CSL_I2cHandle hI2c;
...

/* Define I2C object and HwSetup structure and
   initialize the same */
...
/* I2C Init, Open, HwSetup successfully done in order */
...
inData= 0x65;

status = CSL_i2cWrite(hI2c, & inData);
...
```

9.2.9 CSL_i2cHwSetupRaw

CSL_Status CSL_i2cHwSetupRaw ([CSL_I2cHandle](#) ***hI2c***, [CSL_I2cConfig](#) * ***config***)

Description
This function initializes the device registers with the register-values provided through the Config Data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hI2c        Handle to the I2C

config      Pointer to config structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration is not properly initialized

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling this function.

Post Condition
The registers of the specified I2C instance will be setup according to value passed.

Modifies
Hardware registers of the specified I2C instance.

Example
```
CSL_I2cHandle       hI2c;
CSL_I2cConfig       config = CSL_I2C_CONFIG_DEFAULTS;
CSL_Status          status;
...
status = CSL_i2cHwSetupRaw(hI2c, &config);
...
```

9.2.10 CSL_i2cGetHwStatus

CSL_Status CSL_i2cGetHwStatus ([CSL_I2cHandle](#) ***hI2c***, [CSL_I2cHwStatusQuery](#) ***query***, **void *** ***response***)

Description
This function is used to read the current device configuration, status flags and the value present

associated registers. For various status queries supported and the associated data structure to record the response refer *CSL_I2cHwStatusQuery*. User should allocate memory for the said data type and pass its pointer as an unadorned void* argument to the status query call. .

Arguments

```
hI2c      Handle to the I2C instance

query     The query to this API of I2C which indicates the
          status to be returned.

response  Placeholder to return the status.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware status call is successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - Invalid query command
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_i2cInit()* and *CSL_i2cOpen()* must be called successfully in order before calling *CSL_i2cGetHwStatus()*.

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_I2cHandle          hI2c;
CSL_I2cHwStatusQuery   query = CSL_I2C_QUERY_TX_RDY;
Uint32                 response;
CSL_Status             status;
...

status = CSL_i2cGetHwStatus(hI2c, query, &response);
...
```

9.2.11 CSL_i2cGetBaseAddress

CSL_Status CSL_i2cGetBaseAddress (CSL_InstNum *i2cNum*,
CSL_I2cParam * *pI2cParam*,
CSL_I2cBaseAddress * *pBaseAddress*
)

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_i2cOpen()

function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

`i2cNum`	Specifies the instance of I2C to be opened.
`pI2cParam`	Module specific parameters.
`pBaseAddress`	Pointer to baseaddress structure containing base address details.

Return Value
`CSL_Status`

- `CSL_SOK` - Successful on getting the base address of I2C
- `CSL_ESYS_FAIL` - The instance number is invalid.
- `CSL_ESYS_INVPARAMS` - Invalid Parameter

Pre Condition
None

Post Condition
Base address structure is populated

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status          status;
CSL_I2cBaseAddress  baseAddress;
...
status = CSL_i2cGetBaseAddress(CSL_I2C, NULL, &baseAddress);
...
```

9.3 Data Structures

This section lists the data structures available in the I2C module.

9.3.1 CSL_I2cObj

Detailed Description

This object contains the reference to the instance of I2C opened using the *CSL_i2cOpen()*. The pointer to this is passed to all I2C CSL APIs.

Field Documentation

CSL_InstNum CSL_I2cObj::perNum

This is the instance of I2C being referred by this object

CSL_I2cRegsOvly CSL_I2cObj::regs

The register overlay structure of I2C.

9.3.2 CSL_I2cConfig

Detailed Description

I2C Configuration Structure is used to configure I2C using CSL_i2cHwSetupRaw() function. This is a structure of register values, rather than a structure of register field values like CSL_I2cHwSetup.

Field Documentation

volatile Uint32 CSL_I2cConfig::ICCLKH

I2C Clock High Register

volatile Uint32 CSL_I2cConfig::ICCLKL

I2C Clock Low Register

volatile Uint32 CSL_I2cConfig::ICCNT

I2C Data Count Register

volatile Uint32 CSL_I2cConfig::ICDXR

I2C Data Transmit Register

volatile Uint32 CSL_I2cConfig::ICEMDR

I2C Extended Mode Register

volatile Uint32 CSL_I2cConfig::ICIMR

I2C Interrupt Mask Register

volatile Uint32 CSL_I2cConfig::ICIVR

I2C Interrupt vector register

volatile Uint32 CSL_I2cConfig::ICMDR

I2C Data Receive Register

volatile Uint32 CSL_I2cConfig::ICOAR

I2C Own Address Register

volatile Uint32 CSL_I2cConfig::ICPSC
I2C Pre-scalar Register

volatile Uint32 CSL_I2cConfig::ICSAR
I2C Slave Address Register

volatile Uint32 CSL_I2cConfig::ICSTR
I2C Status Register

9.3.3 CSL_I2cContext

Detailed Description
I2C specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_I2cContext::contextInfo
Context information of I2C. The declaration is just a placeholder for future implementation.

9.3.4 CSL_I2cParam

Detailed Description
I2C specific parameters. Present implementation of I2C CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_I2cParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

9.3.5 CSL_I2cClkSetup

Detailed Description
The clock setup structure has all the fields required to configure the I2C clock.

Field Documentation

Uint32 CSL_I2cClkSetup::clkhighdiv
High time period of the clock

Uint32 CSL_I2cClkSetup::clklowdiv
Low time period of the clock

Uint32 CSL_I2cClkSetup::prescalar
Pre-scalar to the input clock

9.3.6 CSL_I2cHwSetup

Detailed Description
This has all the fields required to configure I2C at Power Up (After a Hardware Reset) or a Soft Reset. This structure is used to setup or obtain existing setup of I2C using *CSL_i2cHwSetup()* and*CSL_i2cGetHwSetup()* functions respectively.

Field Documentation

Uint32 CSL_I2cHwSetup::ackMode
ACK mode while receiver: 0==> ACK Mode, 1==> NACK Mode

Uint32 CSL_I2cHwSetup::addrMode
Addressing Mode :0==> 7-bit Mode, 1==> 10-bit Mode

Uint32 CSL_I2cHwSetup::bcm
I2C Backward Compatibility Mode : 0 ==> Not compatible, 1 ==> Compatible

CSL_I2cClkSetup* CSL_I2cHwSetup::clksetup
Prescalar, Clock Low and Clock High for Clock Setup

Uint32 CSL_I2cHwSetup::dir
Transmitter Mode or Receiver Mode: 1==> Transmitter Mode, 0 ==> Receiver Mode

Uint32 CSL_I2cHwSetup::freeDataFormat
Free Data Format of I2C: 0 ==>Free data format disable, 1 ==> Free data format enable

Uint32 CSL_I2cHwSetup::inten
Interrupt Enable mask The mask can be for one interrupt or OR of multiple interrupts.

Uint32 CSL_I2cHwSetup::loopBackMode
DLBack mode of I2C (master tx-er only): 0 ==> No loopback, 1 ==> Loopback Mode

Uint32 CSL_I2cHwSetup::mode
Master or Slave Mode: 1==> Master Mode, 0==> Slave Mode

Uint32 CSL_I2cHwSetup::ownaddr
Address of the own device

Uint32 CSL_I2cHwSetup::repeatMode
Repeat Mode of I2C: 0==> No repeat mode 1==> Repeat mode

Uint32 CSL_I2cHwSetup::resetMode
I2C Reset Mode: 0==> Reset, 1==> Out of reset

Uint32 CSL_I2cHwSetup::runMode
Run mode of I2C: 0==> No Free Run, 1==> Free Run mode

Uint32 CSL_I2cHwSetup::sttbyteen
Start Byte Mode: 1 ==> Start Byte Mode, 0 ==> Normal Mode

9.3.7 CSL_I2cBaseAddress

Detailed Description
This structure contains the base address information for I2C peripheral instance.

Field Documentation

CSL_I2cRegsOvly CSL_I2cBaseAddress::regs
Base address of the Configuration registers of I2C.

9.4 Enumerations

This section lists the enumerations available in the I2C module.

9.4.1 CSL_I2cHwStatusQuery

enum CSL_I2cHwStatusQuery
Enumeration for queries passed to *CSL_i2cGetHwStatus()*.
This is used to get the status of different operations or to get the existing setup of I2C.

Enumeration values:

CSL_I2C_QUERY_CLOCK_SETUP	To get current clock setup parameters. **Parameters:** *(CSL_I2cClkSetup* *)
CSL_I2C_QUERY_BUS_BUSY	To get the Bus Busy status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_RX_RDY	To get the Receive Ready status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_TX_RDY	To get the Transmit Ready status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_ACS_RDY	To get the Register Ready status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_SCD	To get the Stop Condition Data bit information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_AD0	To get the Address Zero (General Call) detection status. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_RSFULL	To get the Receive overflow status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_XSMT	To get the Transmit underflow status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_AAS	To get the Address as Slave bit information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_AL	To get the Arbitration Lost status information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_RDONE	To get the Reset Done status bit information. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_BITCOUNT	To get number of bits of next byte to be received or

	transmitted. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_INTCODE	To get the interrupt code for the interrupt that occurred. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_SDIR	To get the slave direction. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_NACKSNT	To get the acknowledgement status. **Parameters:** *(Uint32*)*

9.4.2 CSL_I2cHwControlCmd

enum CSL_I2cHwControlCmd
Enumeration for queries passed to *CSL_i2cHwControl()*.
This is used to select the commands to control the operations existing setup of I2C. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_I2C_CMD_ENABLE	Command to enable the I2C module. **Parameters:** *(None)*
CSL_I2C_CMD_RESET	Command to reset the I2C. **Parameters:** *(None)*
CSL_I2C_CMD_OUTOFRESET	Command to make the I2C out of reset. **Parameters:** *(None)*
CSL_I2C_CMD_CLEAR_STATUS	Command to clear the status bits. The argument next to the command specifies the status bit to be cleared. The status bit can be: CSL_I2C_CLEAR_AL, CSL_I2C_CLEAR_NACK, CSL_I2C_CLEAR_ARDY, CSL_I2C_CLEAR_RRDY, CSL_I2C_CLEAR_XRDY, CSL_I2C_CLEAR_GC. **Parameters:** *(None)*
CSL_I2C_CMD_SET_SLAVE_ADDR	Command to set the address of the Slave device. **Parameters:** *(Uint32 *)*
CSL_I2C_CMD_SET_DATA_COUNT	Command to set the Data Count. **Parameters:** *(Uint32 *)*
CSL_I2C_CMD_START	Command to set the start condition. **Parameters:** *(None)*

CSL_I2C_CMD_STOP	Command to set the stop condition. **Parameters:** *(None)*
CSL_I2C_CMD_DIR_TRANSMIT	Command to set the transmission mode. **Parameters:** *(None)*
CSL_I2C_CMD_DIR_RECEIVE	Command to set the receiver mode. **Parameters:** *(None)*
CSL_I2C_CMD_RM_ENABLE	Command to set the Repeat Mode. **Parameters:** *(None)*
CSL_I2C_CMD_RM_DISABLE	Command to disable the Repeat Mode. **Parameters:** *(None)*
CSL_I2C_CMD_DLB_ENABLE	Command to enable the loop back mode. **Parameters:** *(None)*
CSL_I2C_CMD_DLB_DISABLE	Command to disable the loop back mode. **Parameters:** *(None)*

9.5 Macros

#define CSL_I2C_ACK_DISABLE (1)
For enabling the tx of a NACK to the TX-ER, while in the RECEIVER mode

#define CSL_I2C_ACK_ENABLE (0)
For enabling the tx of a ACK to the TX-ER, while in the RECEIVER mode

#define CSL_I2C_ACS_NOT_READY (0)
For indicating that the Access ready signal is low

#define CSL_I2C_ACS_READY (1)
For indicating that the Access ready signal is high

#define CSL_I2C_ADDRSZ_SEVEN (0)
For setting the 7-bit Addressing Mode for I2C

#define CSL_I2C_ADDRSZ_TEN (1)
For setting the 10-bit Addressing Mode

#define CSL_I2C_ARBITRATION_LOST (1)
For indicating Arbitration Lost signal is set

#define CSL_I2C_BCM_DISABLE (0)
For disabling the Backward Compatibility mode of I2C

#define CSL_I2C_BCM_ENABLE (1)
For enabling the Backward Compatibility mode of I2C

#define CSL_I2C_BUS_BUSY (1)
For indicating that the bus is busy

#define CSL_I2C_BUS_NOT_BUSY (0)
For indicating that the bus is not busy

#define CSL_I2C_CLEAR_AL 0x1
Clear the Arbitration Lost status bit

#define CSL_I2C_CLEAR_ARDY 0x4
Clear the Register access ready status bit

#define CSL_I2C_CLEAR_NACK 0x2
Clear the No acknowledge status bit

#define CSL_I2C_CLEAR_RRDY 0x8
Clear the Receive ready status bit

#define CSL_I2C_CLEAR_SCD 0x20
Clear the Stop Condition Detect status bit

#define CSL_I2C_CLEAR_XRDY 0x10
Clear the Transmit ready status bit

**#define CSL_I2C_CONFIG_DEFAULTS **

```
{   \
        CSL_I2C_ICOAR_RESETVAL,     \
        CSL_I2C_ICIMR_RESETVAL,     \
        CSL_I2C_ICSTR_RESETVAL,     \
        CSL_I2C_ICCLKL_RESETVAL,    \
        CSL_I2C_ICCLKH_RESETVAL,    \
        CSL_I2C_ICCNT_RESETVAL,     \
        CSL_I2C_ICSAR_RESETVAL,     \
        CSL_I2C_ICDXR_RESETVAL,     \
        CSL_I2C_ICMDR_RESETVAL,     \
        CSL_I2C_ICIVR_RESETVAL,     \
        CSL_I2C_ICEMDR_RESETVAL,    \
        CSL_I2C_ICPSC_RESETVAL,     \
}
```

Default Values for Config structure

#define CSL_I2C_DIR_RECEIVE (0)
For setting the RECEIVER Mode for I2C

#define CSL_I2C_DIR_TRANSMIT (1)
For setting the TRANSMITTER Mode for I2C

#define CSL_I2C_DLB_DISABLE (0)
For disabling DLB mode of I2C (applicable only in case of MASTER TX-ER)

#define CSL_I2C_DLB_ENABLE (1)
For enabling DLB mode of I2C (applicable only in case of MASTER TX-ER)

#define CSL_I2C_FDF_DISABLE (0)
For disabling the Free Data Format of I2C

#define CSL_I2C_FDF_ENABLE (1)
For enabling the Free Data Format of I2C

#define CSL_I2C_FREE_MODE_DISABLE (0)
For disabling the free run mode of the I2C

#define CSL_I2C_FREE_MODE_ENABLE (1)
For enabling the free run mode of the I2C

#define CSL_I2C_IRS_DISABLE (1)
For taking the I2C out of Reset

#define CSL_I2C_IRS_ENABLE (0)
For putting the I2C in Reset

#define CSL_I2C_MODE_MASTER (1)
For setting the MASTER Mode for I2C

#define CSL_I2C_MODE_SLAVE (0)
For setting the SLAVE Mode for I2C

#define CSL_I2C_RECEIVE_OVERFLOW (1)
For indicating Receive overflow signal is set

#define CSL_I2C_REPEAT_MODE_DISABLE (0)
For disabling the Repeat Mode of the I2C

#define CSL_I2C_REPEAT_MODE_ENABLE (1)
For enabling the Repeat Mode of the I2C

#define CSL_I2C_RESET_DONE (1)
For indicating the completion of Reset

#define CSL_I2C_RESET_NOT_DONE (0)
For indicating the non-completion of Reset

#define CSL_I2C_RX_NOT_READY (0)
For indicating that the Receive ready signal is low

#define CSL_I2C_RX_READY (1)
For indicating that the Receive ready signal is high

#define CSL_I2C_SINGLE_BYTE_DATA (1)
For indicating Single Byte Data signal is set

#define CSL_I2C_STB_DISABLE (0)
For Disabling the Start Byte Mode for I2C(Normal Mode)

#define CSL_I2C_STB_ENABLE (1)
For Enabling the Start Byte Mode for I2C

#define CSL_I2C_TRANSMIT_UNDERFLOW (1)
For indicating Transmit underflow signal is set

#define CSL_I2C_TX_NOT_READY (0)
For indicating that the Transmit ready signal is low

#define CSL_I2C_TX_READY (1)
For indicating that the Transmit ready signal is high

9.6 Typedefs

typedef CSL_I2cObj * CSL_I2cHandle
Handle to the I2C object Handle is used in all accesses to the device parameters.

Chapter 10
INTC Module

Topics

10.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within INTC module.

The CPU has one exception input, one non-maskable interrupt, 12 maskable interrupts, and two dedicated emulation interrupts.The Interrupt Controller supports up to 128 system events.There are 128 system events that act as inputs to the Interrupt Controller. They consist of both internally-generated events (within the megamodule) and chip-level events. In addition to these 128 events, INTC also receives (and routes straight through to the CPU) the non-maskable and reset events.From these event inputs, the Interrupt Controller outputs signals to the CPU:

- One maskable, hardware exception (EXCEP)
- Twelve maskable hardware interrupts (INT4 … INT15)
- One non-maskable signal which can be used as either an interrupt or exception (NMI)
- One reset signal (RESET)

NOTE: The CSL 3.0 INTC module is delivered as a separate library from the remaining CSL modules. When using an embedded operating system that contains interrupt controller/dispatcher support, do not link in the INTC library. For interrupt controller support, DSP/BIOS users should use the HWI (Hardware Interrupt) and ECM (Event Combiner Manager) modules supported under DSP/BIOS v5.21 or later.

10.2 Functions

This section lists the functions available in the INTC module.

10.2.1 CSL_intcInit

CSL_Status CSL_intcInit (CSL_IntcContext * *pContext*)

Description
This is the initialization function for the INTC CSL. This function must be called before calling any other API from this CSL.The context should be initialized such that numEvtEntries is equal to the number of records capable of being held in the eventhandlerRecord.

Arguments

```
pContext       Pointer to module-context structure.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
The context should be initialized such that numEvtEntries is equal to the number of records capable of being held in the eventhandlerRecord.

Post Condition
CPU interrupt table is initialized. Also initializes allocation mask, event offset map and event handler record.

Modifies
None

Example

```
 CSL_IntcContext context;
 CSL_IntcEventHandlerRecord recordTable[10];

 context.numEvtEntries = 10;
 context.eventhandlerRecord = recordTable;

 // Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
    exit (1);
}
...
```

10.2.2 CSL_intcOpen

CSL_IntcHandle CSL_intcOpen (CSL_IntcObj * *intcObj*,

CSL_IntcEventId	*eventId*,
CSL_IntcParam *	*param*,
CSL_Status *	*status*

)

Description

The API would reserve an interrupt-event for use. It returns a valid handle to the event only if the event is not currently allocated. The user could release the event after use by calling CSL_intcClose(). The CSL-object ('intcObj') that the user passes would be used to store information pertaining handle.

Arguments

`intcObj`	`Pointer to the CSL-object allocated by the user`
`eventId`	`The event-id of the interrupt`
`param`	`Pointer to the Intc specific parameter`
`status`	`Pointer for returning status of the function call`

Return Value

`CSL_IntcHandle`

- Valid INTC handle identifying the event

Pre Condition

The INTC must be successfully initialized via CSL_intcInit() before calling this function.

Post Condition

1. INTC object structure is populated
2. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid intc handle is returned
- `CSL_ESYS_FAIL` - The open failed
- `CSL_INTC_BADHANDLE` – Invalid handle

Modifies

1. The status variable
2. INTC object structure

Example:

```
CSL_IntcObj                 intcObj20;
CSL_IntcHandle              hIntc20;
CSL_IntcGlobalEnableState state;

CSL_IntcContext            context;
CSL_Status                 intStat;
CSL_IntcParam              vectId;

context.numEvtEntries = 0;
```

```
context.eventhandlerRecord = NULL;

// Init Module
CSL_intcInit(&context);

// NMI Enable
CSL_intcGlobalNmiEnable();

// Enable Global Interrupts
intStat = CSL_intcGlobalEnable(&state);

// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20, CSL_INTC_EVENTID_RIOINT0,
                       &vectId,
                       NULL);

// Close handle
CSL_intcClose(hIntc20);
...
```

10.2.3 CSL_intcClose

CSL_Status CSL_intcClose (CSL_IntcHandle *hIntc*)

Description
This intc handle can no longer be used to access the event. The event is de-allocated and further access to the event resources are possible only after opening the event object again.

Arguments

hIntc	Handle identifying the event

Return Value
CSL_Status

- CSL_SOK - Close successful
- CSL_INTC_BADHANDLE - The handle passed is invalid

Pre Condition
Functions CSL_intcInit() and CSL_intcOpen() have to be called in that order successfully before calling this function.

Post Condition
1. CPU interrupt could be used again
2. The intc CSL APIs can not be called until the intc CSL is reopened again using CSL_intcOpen().

Modifies
CSL_intcObj structure values

Example

```
CSL_IntcContext              context;
CSL_Status                   intStat;
CSL_IntcParam                vectId;
CSL_IntcObj                  intcObj20;
CSL_IntcHandle               hIntc20;
CSL_IntcEventHandlerRecord   recordTable[10];

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
   exit (1);
}

// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20,
                       CSL_INTC_EVENTID_RIOINT0,
                       &vectId, \
                       NULL);

// Close handle
intStat = CSL_intcClose(hIntc20);
...
```

10.2.4 CSL_intcPlugEventHandler

CSL_Status CSL_intcPlugEventHandler(CSL_IntcHandle *hIntc*,
CSL_IntcEventHandlerRecord * *eventHandlerRecord*
)

Description
Associate an event-handler with an event CSL_intcPlugEventHandler(..) ties an event-handler to an event; so that the occurence of the event, would result in the event-handler being invoked.

Arguments

```
hIntc                 Handle identifying the interrupt-event

eventHandlerRecord    Provides the details of the event-handler
```

Return Value CSL_Status
- CSL_SOK - Succussful completion of PlugEventHandler
- CSL_ESYS_FAIL - Non completion of PlugEventHandler

Pre Condition
Functions CSL_intcInit() and CSL_intcOpen() must be called in order successfully before calling this function.

Post Condition
None

Modifies
Event Handler Record structure values

Example:

```
    CSL_IntcObj                    intcObj20;
    CSL_IntcHandle                 hIntc20;
    CSL_IntcGlobalEnableState      state;
    CSL_IntcEventHandlerRecord     EventRecord;
    CSL_IntcContext                context;
    CSL_Status                     intStat;
    CSL_IntcParam                  vectId;
    CSL_Status                     status;

    context.numEvtEntries = 0;
    context.eventhandlerRecord = NULL;

    // Init Module
    CSL_intcInit(&context);

    // NMI Enable
    CSL_intcGlobalNmiEnable();

    // Enable Global Interrupts
    intStat = CSL_intcGlobalEnable(&state);

    // Opening a handle for the Event 20 at vector id 4
    vectId = CSL_INTC_VECTID_4;
    hIntc20 = CSL_intcOpen(&intcObj20, CSL_INTC_EVENTID_RIOINT0,
                           &vectId ,
                           NULL);

    EventRecord.handler = &event20Handler;
    EventRecord.arg = hIntc20;
    Status = CSL_intcPlugEventHandler(hIntc20,&EventRecord);


    // Close handle
    CSL_intcClose(hIntc20);
    ...
   }
  void event20Handler( CSL_IntcHandle hIntc)
  {
    ...
  }
```

10.2.5 CSL_intcHookIsr

CSL_Status CSL_intcHookIsr	**(**	[CSL_IntcVectId]()	***evtId*,**
		void *	***isrAddr***

)

Description

Hook up an exception handler This API hooks up the handler to the specified exception. Note: In this case, it is done by inserting a B(ranch) instruction to the handler. Because of the restriction in the instruction, the handler must be within 32MB of the exception vector. In addition, the function assumes that the exception vector table is located at its default ("low") address.

Arguments

```
evtId       Interrupt Vector identifier

isrAddr     Pointer to the handler
```

Return Value

`CSL_Status`

- `CSL_SOK` - CSL_intcHookIsr Successful

Pre Condition

The function CSL_intcInit() has to be called successfully before calling this function.

Post Condition

Hooks up the handler to the specified exception

Modifies

None

Example:

```
CSL_IntcContext             context;
CSL_Status                  intStat;
CSL_IntcParam               vectId;
CSL_IntcObj                 intcObj20;
CSL_IntcHandle              hIntc20;
CSL_IntcDropStatus          drop;
CSL_IntcEventHandlerRecord  recordTable[10];
CSL_IntcGlobalEnableState   state;
Uint32                      intrStat;

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK)
   exit (1);
// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20,
                       CSL_INTC_EVENTID_RIOINT0,
                       &vectId ,
```

```
                              NULL);

            CSL_intcGlobalNmiEnable();

            // Enable Global Interrupts
            intStat = CSL_intcGlobalEnable(&state);

            // Hook Isr appropriately
            CSL_intcHookIsr(CSL_INTC_VECTID_4,&isrVect4);
                ...
            }
            interrupt void isrVect4() {
                ...
            }
```

10.2.6 CSL_intcHwControl

CSL_Status CSL_intcHwControl	**(**	**CSL_IntcHandle**	***hIntc*,**
		CSL_IntcHwControlCmd	***controlCommand*,**
		void *	***commandArg***
	)		

Description
This API perform a control-operation. This API is used to invoke any of the supported control-operations supported by the module.

Arguments

`hIntc`	`Handle identifying the event`
`controlCommand`	`The command to this API indicates the action to be taken on INTC.`
`commandArg`	`An optional argument.`

Return Value
`CSL_Status`

- `CSL_SOK` - HwControl successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command

Pre Condition
Functions CSL_intcInit() and CSL_intcOpen() must be called in order successfully before calling this function.

Post Condition
INTC registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
The hardware registers of INTC.

Example

```
CSL_IntcObj                 intcObj20;
CSL_IntcHandle              hIntc20;
CSL_IntcGlobalEnableState state;
CSL_IntcContext             context;
CSL_Status                  intStat;
CSL_IntcParam               vectId;

context.numEvtEntries = 0;
context.eventhandlerRecord = NULL;

// Init Module
CSL_intcInit(&context);

// NMI Enable
CSL_intcGlobalNmiEnable();

// Enable Global Interrupts
intStat = CSL_intcGlobalEnable(&state);

// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20, CSL_INTC_EVENTID_RIOINT0,
                       &vectId,NULL);

CSL_intcHwControl(hIntc20,CSL_INTC_CMD_EVTENABLE,NULL);
...
```

10.2.7 CSL_intcGetHwStatus

CSL_Status CSL_intcGetHwStatus	**(**	CSL_IntcHandle	***hIntc*,**
		CSL_IntcHwStatusQuery	***myQuery*,**
		void *	***answer***
	)		

Description
Queries the peripheral for status. The CSL_intcGetHwStatus(..) API could be used to retrieve status or configuration information from the peripheral. The user must allocate an object that would hold the retrieved information and pass a pointer to it to the function. The type of the object is specific to the query-command.

Arguments

```
hIntc           Handle identifying the event

myQuery         The query to this API of INTC which indicates the
                status to be returned.
```

```
    answer          Placeholder to return the status.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Getting the status of INTC is successful
- `CSL_ESYS_INVQUERY` - The query passed is invalid
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
The functions CSL_intcInit(), CSL_intcOpen() must be called successfully in that order before this API can be invoked.

Post Condition
None

Modifies
Third parameter, answer value

Example:

```
CSL_IntcContext             context;
CSL_Status                  intStat;
CSL_IntcParam               vectId;
CSL_IntcObj                 intcObj20;
CSL_IntcHandle              hIntc20;
CSL_IntcEventHandlerRecord  recordTable[10];
CSL_IntcGlobalEnableState   state;
Uint32                      intrStat;

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK)
   exit (1);;
// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20,
                       CSL_INTC_EVENTID_RIOINT0,
                       &vectId ,
                        NULL);

// NMI Enable
CSL_intcGlobalNmiEnable();

// Enable Global Interrupts
intStat = CSL_intcGlobalEnable(&state);

do {
    CSL_intcGetHwStatus(hIntc20,CSL_INTC_QUERY_PENDSTATUS,\
                        (void*)&intrStat);
```

```
} while (!intrStat);

// Close handle
CSL_intcClose(hIntc20);
...
```

10.2.8 CSL_intcGlobalEnable

CSL_Status CSL_intcGlobalEnable (CSL_IntcGlobalEnableState * *prevState*)

Description
Globally enable interrupts. The API enables the global interrupt by manipulating the processor's global interrupt enable/disable flag. If the user wishes to restore the enable-state at a later point, they may store the current state using the parameter, which could be used with CSL_intcGlobalRestore(..). CSL_intcGlobalEnable(..) must be called from a privileged mode.

Arguments

```
prevState    Pointer to object that would store current
             stateObject that contains information about previous
             state
```

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() has to be called successfully before calling this function.

Post Condition
Enables interrupts globally

Modifies
CPU registers

Example:

```
CSL_Status          status;
...
status = CSL_intcGlobalEnable(NULL);
...
```

10.2.9 CSL_intcGlobalDisable

CSL_Status CSL_intcGlobalDisable (CSL_IntcGlobalEnableState * *prevState*)

Description
Globally disable interrupts. The API disables the global interrupt by manipulating the processor's global interrupt enable/disable flag. If the user wishes to restore the enable-state at a later point, they may store the current state using the parameter, which could be used with CSL_intcGlobalRestore(..). CSL_intcGlobalDisable(..) must be called from a privileged mode.

Arguments

```
prevState    Pointer to object that would store current
             stateObject that contains information about previous
             state
```

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() has to be called successfully before calling this function.

Post Condition
Disables interrupts globally

Modifies
CPU registers

Example:

```
CSL_Status          status;
...
status = CSL_intcGlobalDisable(NULL);
...
```

10.2.10 CSL_intcGlobalRestore

CSL_Status CSL_intcGlobalRestore (CSL_IntcGlobalEnableState *prevState*)

Description
Restores global interrupt enable/disable to a previous state. The API restores the global interrupt enable/disable state to a previous state as recorded by the global-event-enable state passed as an argument. CSL_intcGlobalRestore(..) must be called from a privileged mode.

Arguments

```
prevState   Object containing information about previous state
```

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() has to be called successfully before calling this function

Post Condition
Restores global interrupt enable/disable to a previous state

Modifies
None

Example:

```
CSL_Status                    status;
CSL_IntcGlobalEnableState     prevState;
...
status = CSL_intcGlobalRestore(prevState);
...
```

10.2.11 CSL_intcGlobalNmiEnable

CSL_Status CSL_intcGlobalNmiEnable (void)

Description
This API enables global NMI.

Arguments
None

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
Global NMI is enabled

Modifies
CPU registers

Example:

```
CSL_Status          status;
...
status = CSL_intcGlobalNmiEnable();
...
```

10.2.12 CSL_intcGlobalExcepEnable

CSL_Status CSL_intcGlobalExcepEnable (void)

Description
This API enables global exception.

Arguments
None

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition

The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
Global exception is enabled

Modifies
CPU registers

Example:
```
CSL_Status              status;
...

status = CSL_intcGlobalExcepEnable();
...
```

10.2.13 CSL_intcGlobalExtExcepEnable

CSL_Status CSL_intcGlobalExtExcepEnable (void)

Description
This API enables external exception.

Arguments
None

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
External exception is enabled

Modifies
CPU registers

Example:
```
CSL_Status              status;
...

status = CSL_intcGlobalExtExcepEnable();
...
```

10.2.14 CSL_intcGlobalExcepClear

CSL_Status CSL_intcGlobalExcepClear ([CSL_IntcExcep](#) *exc*)

Description
This API clears Global Exceptions.

Arguments

```
exc      Exception to be cleared NMI/SW/EXT/INT
```

Return Value
`CSL_Status`

- `CSL_SOK` on success

Pre Condition
The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
Global exception is cleared

Modifies
CPU registers

Example:
```
CSL_Status       status;
CSL_IntcExcep    exc;
...

status = CSL_intcGlobalExcepClear(exc);
...
```

10.2.15 CSL_intcExcepAllEnable

CSL_Status CSL_intcExcepAllEnable **(** **CSL_IntcExcepEn** ***excepMask***, **CSL_BitMask32** ***excVal***, **CSL_BitMask32 *** ***prevState*** **)**

Description
This API enables all exceptions.

Arguments

```
excepMask    Exception to be cleared NMI/SW/EXT/INT
excVal       Event Value
prevState    Pointer to Pre state information
```

Return Value
`CSL_Status`

- `CSL_SOK` - on success
- `CSL_ESYS_INVPARAMS – Invalid parameters`

Pre Condition

The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcContext             context;
CSL_Status                  intcStat;
CSL_IntcEventHandlerRecord  recordTable[10];
CSL_BitMask32               prevState;

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
   exit (1);
// Enable exception events 9,10,11.
intcStat = CSL_intcExcepAllEnable(CSL_INTC_EXCEP_0TO31,
                                  0x0F00,&prevState);
...
```

10.2.16 CSL_intcExcepAllDisable

CSL_Status CSL_intcExcepAllDisable	**(**	CSL_IntcExcepEn	***excepMask*,**
		CSL_BitMask32	***excVal*,**
		CSL_BitMask32 *****	***prevState***
	)		

Description
This API disables all exceptions.

Arguments

```
excepMask    Exception Mask

excVal       Event Value

prevState    Previous state information
```

Return Value
`CSL_Status`

- CSL_SOK on success
- `CSL_ESYS_INVPARAMS – Invalid parameters`

Pre Condition

The function CSL_intcInit() must be called successfully before calling this function..

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcContext              context;
CSL_Status                   intcStat;
CSL_IntcEventHandlerRecord   recordTable[10];
CSL_BitMask32                prevState;

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
   exit (1);
// Enable exception events 9,10,11.
intcStat = CSL_intcExcepAllDisable(CSL_INTC_EXCEP_0TO31, \
                                   0x0F00,&prevState);
...
```

10.2.17 CSL_intcExcepAllRestore

CSL_Status CSL_intcExcepAllRestore **(** CSL_IntcExcepEn ***excepMask*****,**
CSL_IntcGlobalEnableState ***prevState***
)

Description
This API restores all exceptions.

Arguments

```
excepMask    Exception Mask

prevState    BitMask to be restored
```

Return Value
`CSL_Status`

- CSL_SOK on success
- `CSL_ESYS_INVPARAMS – Invalid parameters`

Pre Condition
The function CSL_intcInit() must be called successfully before calling this function.

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcContext              context;
CSL_Status                   intcStat;
CSL_IntcEventHandlerRecord   recordTable[10];
CSL_IntcGlobalEnableState    prevState;

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
    exit (1);
intcStat = CSL_intcExcepAllDisable(CSL_INTC_EXCEP_0TO31,0x0F00, \
&prevState);

// Restore
intcStat = CSL_intcExcepAllRestore(CSL_INTC_EXCEP_0TO31,
prevState);
...
```

10.2.18 CSL_intcExcepAllClear

CSL_Status CSL_intcExcepAllClear (CSL_IntcExcepEn *excepMask*, CSL_BitMask32 *excVal*)

Description
This clears the exception flags.

Arguments

```
excepMask       Exception Mask

excVal          Holder for the event bitmask to be cleared
```

Return Value
`CSL_Status`

- `CSL_SOK` - Intc Excep All Clear return successful
- `CSL_ESYS_INVPARAMS – Invalid parameters`

Pre Condition
CSL_intcInit() and CSL_intcExcepAllEnable() must be called before using this API.

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcContext context;
CSL_Status intcStat;
CSL_IntcEventHandlerRecord recordTable[10];

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
   exit (1);
// Clear exception events 9,10,11.

intcStat = CSL_intcExcepAllClear(CSL_INTC_EXCEP_0TO31,0x0F00);
...
```

10.2.19 CSL_intcExcepAllStatus

CSL_Status CSL_intcExcepAllStatus	**(**	**CSL_IntcExcepEn**	***excepMask*,**
		CSL_BitMask32 *	***status***
	)		

Description
This obtains the status of the exception flags.

Arguments

`excepMask`	`Exception Mask`
`status`	`Holder for the event bitmask to be cleared`

Return Value
`CSL_Status`

- `CSL_SOK` - intc Excep All Status return successful
- `CSL_ESYS_INVPARAMS – Invalid parameters`

Pre Condition
CSL_intcInit() must be called before using this API.

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcContext             context;
CSL_Status                  intcStat;
CSL_BitMask32               exp0Stat;
CSL_IntcEventHandlerRecord  recordTable[10];

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;

// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK) {
   exit (1);
intcStat = CSL_intcExcepAllStatus(CSL_INTC_EXCEP_0TO31,&exp0Stat);
...
```

10.2.20 CSL_intcQueryDropStatus

CSL_Status CSL_intcQueryDropStatus (CSL_IntcDropStatus * *drop*)

Description
Queries the peripheral for Drop status. The CSL_intcQueryDropStatus(..) API could be used to retrieve drop status.

Arguments

```
drop            Pointer to drop status structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful
- `CSL_ESYS_INVPARAMS` - Invalid Parameter

Pre Condition
CSL_intcInit(), CSL_intcOpen() must be invoked before this call.

Post Condition
None

Modifies
None

Example:

```
CSL_IntcContext             context;
CSL_IntcParam               vectId;
CSL_IntcObj                 intcObj20;
CSL_IntcHandle              hIntc20;
CSL_IntcDropStatus          drop;
CSL_IntcEventHandlerRecord  recordTable[10];

context.numEvtEntries = 10;
context.eventhandlerRecord = recordTable;
```

```
// Init Module
...
if (CSL_intcInit(&context) != CSL_SOK)
   exit (1);
// Opening a handle for the Event 20 at vector id 4

vectId = CSL_INTC_VECTID_4;
hIntc20 = CSL_intcOpen(&intcObj20,
                       CSL_INTC_EVENTID_RIOINT0,
                       &vectId ,
                       NULL);

// Drop Enable
CSL_intcHwControl(hIntc20,CSL_INTC_CMD_EVTDROPENABLE,NULL);
// Query Drop status
CSL_intcQueryDropStatus(&drop);

// Close handle
CSL_intcClose(hIntc20);
...
```

10.2.21 CSL_intcMapEventVector

void CSL_intcMapEventVector **(** **CSL_IntcEventId** ***eventId*** **CSL_IntcVectId** ***vectId*** **)**

Description
This API Maps the event to the given CPU vector.

Arguments

```
eventId     Intc event Identifier

vectId      Intc vector identifier
```

Return Value
None

Pre Condition
CSL_intcInit() must be invoked before this call.

Post Condition
Maps the event to the given CPU vector

Modifies
INTC hardware registers

Example:

```
CSL_IntcVectId              vectId;
CSL_IntcEventId             eventId;
...
CSL_intcMapEventVector(eventId, vectId);
...
```

10.2.22 CSL_intcEventEnable

CSL_IntcEventEnableState CSL_intcEventEnable(CSL_IntcEventId *eventId*)

Description
This API enables particular event (EVTMASK0/1/2/3 bit programmation).

Arguments
```
eventId  -  Intc event Identifier
```

Return Value
CSL_IntcEventEnableState - previous state

Pre Condition
None

Post Condition
Particular event will be enabled

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
CSL_IntcEventEnableState    prevState;
...
prevState = CSL_intcEventEnable(eventId);
...
```

10.2.23 CSL_intcEventDisable

CSL_IntcEventEnableState CSL_intcEventDisable(CSL_IntcEventId *eventId*)

Description
This API disable particular event (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
```

Return Value
CSL_IntcEventEnableState – previous state.

Pre Condition
None

Post Condition
Particular event will be disabled

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
CSL_IntcEventEnableState    eventStat;
...
eventStat = CSL_intcEventDisable(eventId);
...
```

10.2.24 CSL_intcEventRestore

void CSL_intcEventRestore **(CSL_IntcEventId** ***eventId*** **CSL_InctEventEnableState** ***restoreVal***)

Description
This API restores particular event (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
restoreVal  Restore value
```

Return Value
None

Pre Condition
None

Post Condition
Particular event will be restored

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId          eventId;
CSL_IntcEventEnableState restoreVal;
CSL_IntcEventEnableState eventStat;
...
eventStat = CSL_intcEventResore(eventId, restoreVal);
...
```

10.2.25 CSL_intcEventSet

void CSL_intcEventSet **(CSL_IntcEventId** ***eventId*** **)**

Description
This API sets event (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
```

Return Value
None

Pre Condition
None

Post Condition
Particular event will set

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId     eventId;
...
CSL_intcEventSet(eventId);
...
```

10.2.26 CSL_intcEventClear

void CSL_intcEventClear (CSL_IntcEventId *eventId*)

Description
This API clears event (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
```

Return Value
None

Pre Condition
None

Post Condition
Particular event will be cleared

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId     eventId;
```

```
...
CSL_intcEventClear(eventId);
...
```

10.2.27 CSL_intcCombinedEventClear

void CSL_intcCombinedEventClear (CSL_IntcEventId *eventId*
CSL_BitMask32 *clearMask*
)

Description
This API clears particular combined events (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
clearMask   Bit mask events to be cleared
```

Return Value
None

Pre Condition
None

Post Condition
Particular combined events will be cleared

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId    eventId;
CSL_BitMask32      clearMask;
...
CSL_intcEventClear(eventId, clearMask);
...
```

10.2.28 CSL_intcCombinedEventGet

CSL_BitMask32 CSL_intcCombinedEventGet (CSL_IntcEventId *eventId*)

Description
This API gets particular combined events (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId     Intc event Identifier
```

Return Value
CSL_BitMask32 – The combined event information

Pre Condition

None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcEventId    eventId;
CSL_BitMask32      combEvtStat;
...
combEvtStat = CSL_intcEventClear(eventId);
```

10.2.29 CSL_intcCombinedEventEnable

**CSL_BitMask32 CSL_intcCombinedEventEnable(CSL_IntcEventId *eventId*
CSL_BitMask32 *enableMask*
)**

Description
This API enables particular combined events (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId      Intc event Identifier
enableMask   Bit mask events to be enabled
```

Return Value
CSL_BitMask32 - previous state.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId    eventId;
CSL_BitMask32      enableMask;
CSL_BitMask32      combEvtStat;
...
combEvtStat = CSL_intCombinedEventEnable(eventId, enableMask);
...
```

10.2.30 CSL_intcCombinedEventDisable

CSL_BitMask32 CSL_intcCombinedEventDisable(CSL_IntcEventId ***eventId***
CSL_BitMask32 ***disableMask***
)

Description
This API disables particular combined events (EVTMASK0/1/2/3 bit programmation).

Arguments

```
eventId       Intc event Identifier
disableMask   Bit mask events to be disabled
```

Return Value
CSL_BitMask32 - previous state.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId    eventId;
CSL_BitMask32      disableMask;
CSL_BitMask32      combEvtStat;
...
combEvtStat=CSL_intCombinedEventDisable(eventId, disableMask);
...
```

10.2.31 CSL_intcCombinedEventRestore

void CSL_intcCombinedEventRestore (CSL_IntcEventId ***eventId***
CSL_BitMask32 ***restoreMask***
)

Description
This API restores particular combined events

Arguments

```
eventId       Intc event Identifier
restoreMask   Bit mask events to be restored
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId    eventId;
CSL_BitMask32      restoreMask;
CSL_BitMask32      combEvtStat;
...
combEvtStat=CSL_intCombinedEventRestore(eventId, restoreMask);
...
```

10.2.32 CSL_intcInterruptDropEnable

void CSL_intcInterruptDropEnable (CSL_BitMask32 *dropMask*)

Description
This API enables interrupts for which drop detection .

Arguments

```
dropMask – Vectorid Mask
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_BitMask32      dropMask;
...
CSL_intcInterruptDropEnable(dropMask );
...
```

10.2.33 CSL_intcInterruptDropDisable

void CSL_intcInterruptDropDisable (CSL_BitMask32 *dropMask*)

Description
This API disables interrupts for which drop detection.

Arguments

```
dropMask – Vectorid Mask
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_BitMask32      dropMask;
...
CSL_intcInterruptDropEnable(dropMask);
...
```

10.2.34 CSL_intcInvokeEventHandle

CSL_Status CSL_intcInvokeEventHandle (CSL_IntcEventId *evtId*)

Description
This API is for the purpose of exception handler which will need to be written by the user. This API invokes the event handler registered by the user at the time of event open and event handler registration.

Arguments

```
evtId – Intc event identifier
```

Return Value
CSL_SOK – success.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_Status            status;
CSL_IntcEventId       evtId;
...
status = CSL_intcInvokeEventHandle(evtId);
...
```

10.2.35 CSL_intcQueryEventStatus

Bool CSL_intcQuerryEventStatus (CSL_IntcEventId *eventId*)

Description
This API is used to check whether the specied event is enabled or not

Arguments
`eventId – Intc event identifier.`

Return Value
Bool

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcEventId       eventId;
Bool                  returnVal;
...
returnVal = CSL_intcQueryEventStatus(eventId);
...
```

10.2.36 CSL_intcInterruptEnable

Uint32 CSL_intcInterruptEnable (CSL_IntcVectId *vetctId*)

Description
This API is used to enable the interrupt

Arguments
`vectId – vector id to enable.`

Return Value
Uint32 - previous state.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId         vectId;
Uint32                 returnVal;
...
returnVal = CSL_intcInterruptEnable(vectId);
...
```

10.2.37 CSL_intcInterruptDisable

Uint32 CSL_intcInterruptDisable (CSL_IntcVectId *vetctId*)

Description
This API is used to disable the interrupt

Arguments
`vectId – vector id to disable.`

Return Value
Uint32 - previous state.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId         vectId;
Uint32                 returnVal;
...
returnVal = CSL_intcInterruptDisable(vectID);
...
```

10.2.38 CSL_intcInterruptRestore

void CSL_intcInterruptRestore (CSL_IntcVectId *vetctId*)
Uint32 *restoreVal*
)

Description
This API restores the interrupt.

Arguments
    `vectId – vector id to restore.`
    `restoreVal – Restore value`

Return Value
None.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId        vectId;
Uint32                restoreVal;
...
CSL_intcInterruptDisable(vectId, restoreVal);
...
```

10.2.39 CSL_intcInterruptSet

void CSL_intcInterruptSet (CSL_IntcVectId *vetctId*)

Description
This API sets the interrupt.

Arguments
    `vectId – Vector id to set.`

Return Value
None.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId        vectId;
...
```

```
CSL_intcInterruptSet(vectId);
...
```

10.2.40 CSL_intcInterruptClear

void CSL_intcInterruptClear (CSL_IntcVectId *vetctId*)

Description
This API clears the specified interrupt.

Arguments
`vectId – Vector id to clear.`

Return Value
None.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId        vectId;
...
CSL_intcInterruptClear(vectId);
...
```

10.2.41 CSL_intcQueryInterruptStatus

Bool CSL_intcQuerryInterruptStatus (CSL_IntcVectId *vetctId*)

Description
This API is to check whether a specified CPU interrupt is pending or not.

Arguments
`vectId – Vector id.`

Return Value
Bool.

Pre Condition
None

Post Condition
None

Modifies
None

Example:

```
CSL_IntcVectId          vectId;
Bool                    returnVal;
...
returnVal = CSL_intcInterruptSet(vectId);
...
```

10.2.42 CSL_intcExcepEnable

CSL_IntcEventEnableState CSL_intcExcepEnable(CSL_IntcEventId *eventId*)

Description
This API enables the specified exception event.

Arguments
`eventId – exception event id to be enabled.`

Return Value
CSL_IntcEventEnableState – old state.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
CSL_IntcEventEnableState    returnVal;
...
returnVal = CSL_intcExcepEnable(eventId);
...
```

10.2.43 CSL_intcExcepDisable

CSL_IntcEventEnableState CSL_intcExcepDisable(CSL_IntcEventId *eventId*)

Description
This API disables the specified exception event..

Arguments
`eventId – exception event id to be disabled`

Return Value
CSL_IntcEventEnableState – old state.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
CSL_IntcEventEnableState    returnVal;
...
returnVal = CSL_intcExcepDisable(eventId);
...
```

10.2.44 CSL_intcExcepRestore

void CSL_intcExcepRestore **(CSL_IntcEventId** ***eventId*** **Uint32** ***restoreVal*** **)**

Description
This API restores the specified exception event.

Arguments

```
eventId    – exception event id to be restored.

restoreVal – restore value.
```

Return Value
None.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
Uint32                      restoreVal;
...
CSL_intcExcepRestore(eventId, restoreVal);
...
```

10.2.45 CSL_intcExcepClear

void CSL_intcInterruptSet (CSL_IntcEventId *eventId*)

Description
This API enables the specified exception event.

Arguments
```
eventId – exception event id to be cleared.
```

Return Value
None.

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_IntcEventId             eventId;
...
CSL_intcExcepClear(eventId);
...
```

10.3 Data Structures

This section lists the data structures available in the INTC module.

10.3.1 CSL_IntcObj

Detailed description
The interrupt handle object. This object is used referenced by the handle to identify the event.

Field Documentation

CSL_IntcObj::eventId
The event-id

CSL_IntcVectId CSL_IntcObj::vectId
The vector-id

10.3.2 CSL_IntcContext

Detailed description
INTC Module Context

Field Documentation

CSL_BitMask32 CSL_IntcContext::eventAllocMask[(CSL_INTC_EVENTID_CNT + 31) / 32]
Event allocation mask

CSL_IntcEventHandlerRecord* CSL_IntcContext::eventhandlerRecord
Pointer to the event handle record

Uint16 CSL_IntcContext::numEvtEntries
Number of event entries

Int8 CSL_IntcContext::offsetResv[128]
Reserved

10.3.3 CSL_IntcEventHandlerRecord

Detailed description
Event Handler Record. Used to setup the event-handler using CSL_intcPlugEventHandler(..)

Field Documentation

void* CSL_IntcEventHandlerRecord::arg
The argument to be passed to the handler when it is invoked.

CSL_IntcEventHandler CSL_IntcEventHandlerRecord::handler
Pointer to the event handler

10.3.4 CSL_IntcDropStatus

Detailed description
The drop status structure. This object is used along with the CSL_intcQueryDropStatus() API.

Field Documentation

Bool CSL_IntcDropStatus::drop
Whether dropped/not

CSL_IntcEventId CSL_IntcDropStatus::eventId
The event-id

CSL_IntcVectId CSL_IntcDropStatus::vectId
The vect-id

10.4 Enumerations

This section lists the enumerations available in the INTC module.

10.4.1 CSL_IntcVectId

enum CSL_IntcVectId
Interrupt Vector Ids

Enumeration values:

CSL_INTC_VECTID_NMI	Should be used only along with CSL_intcHookIsr()
CSL_INTC_VECTID_4	CPU Vector 4
CSL_INTC_VECTID_5	CPU Vector 5
CSL_INTC_VECTID_6	CPU Vector 6
CSL_INTC_VECTID_7	CPU Vector 7
CSL_INTC_VECTID_8	CPU Vector 8
CSL_INTC_VECTID_9	CPU Vector 9
CSL_INTC_VECTID_10	CPU Vector 10
CSL_INTC_VECTID_11	CPU Vector 11
CSL_INTC_VECTID_12	CPU Vector 12
CSL_INTC_VECTID_13	CPU Vector 13
CSL_INTC_VECTID_14	CPU Vector 14
CSL_INTC_VECTID_15	CPU Vector 15
CSL_INTC_VECTID_COMBINE	Should be used at the time of opening an Event handle to specify that the event needs to go to the combiner
CSL_INTC_VECTID_EXCEP	Should be used at the time of opening an Event handle to specify that the event needs to go to the exception combiner.

10.4.2 CSL_IntcHwControlCmd

enum CSL_IntcHwControlCmd
Enumeration of the control commands
These are the control commands that could be used with CSL_intcHwControl(..). Some of the commands expect an argument as documented along side the description of the command.

Enumeration values:

CSL_INTC_CMD_EVTDISABLE	Disables the event. The parameter should be 1. "*CSL_BitMask32" for Combined events* *2. "None" for other evnts.* **Parameters:** *(CSL_BitMask32 *)*
CSL_INTC_CMD_EVTSET	Sets the event manually. **Parameters:** *None*
CSL_INTC_CMD_EVTCLEAR	Clears the event (if pending). The parameter should be

	1. "*CSL_BitMask32" for Combined events* *2. "None" for other evnts* **Parameters:** *(CSL_BitMask32 *)*
CSL_INTC_CMD_EVTDROPENABLE	Enables the Drop Event detection feature for this event. **Parameters:** *None*
CSL_INTC_CMD_EVTDROPDISABLE	Disables the Drop Event detection feature for this event. **Parameters:** *None*
CSL_INTC_CMD_EVTINVOKEFUNCTION	To be used ONLY to invoke the associated Function handlewith Event when the user is writing an exception handling routine. **Parameters:** *None*
CSL_INTC_CMD_EVTENABLE	Enables the event. The parameter should be 1. "*CSL_BitMask32" for Combined events* *2. "None" for other evnts* **Parameters:** *(CSL_BitMask32 *)*

10.4.3 CSL_IntcHwStatusQuery

enum CSL_IntcHwStatusQuery
Enumeration of the queries. These are the queries that could be used with CSL_intcGetHwStatus(..). The queries return a value through the object pointed to by the pointer that it takes as an argument. The argument supported by the query is documented along side the description of the query.
Enumeration values:

CSL_INTC_QUERY_PENDSTATUS	The Pend Status of the Event is queried. **Parameters:** *Bool*

10.4.4 CSL_IntcExcepEn

enum CSL_IntcExcepEn
Enumeration of the exception mask registers. These are the symbols used along with the value to be programmed into the Exception mask register.

Enumeration values:

CSL_INTC_EXCEP_0TO31	Symbol for EXPMASK[0]. Parameters: *BitMask* for EXPMASK0
CSL_INTC_EXCEP_32TO63	Symbol for EXPMASK[1].

Parameters:
BitMask for EXPMASK1

CSL_INTC_EXCEP_64TO95 Symbol for EXPMASK[2].
Parameters:
BitMask for EXPMASK2

CSL_INTC_EXCEP_96TO127 Symbol for EXPMASK[3].
Parameters:
BitMask for EXPMASK3

10.4.5 CSL_IntcExcep

enum CSL_IntcExcep
Enumeration of the exception
These are the symbols used along with the Exception Clear API.

Enumeration values:

CSL_INTC_EXCEPTION_NMI Symbol for NMI.
Parameters:
None

CSL_INTC_EXCEPTION_EXT Symbol for External Exception.
Parameters:
None

CSL_INTC_EXCEPTION_INT Symbol for Internal Exception.
Parameters:
None

CSL_INTC_EXCEPTION_SW Symbol for Software Exception
Parameters:
None

10.5 Macros

#define CSL_INTC_BADHANDLE (0)
Invalid handle

#define CSL_INTC_EVENTID_CNT 128
Number of Events in the System

#define CSL_INTC_EVTHANDLER_NONE ((CSL_IntcEventHandler) 0)
Indicates there is no associated event-handler

#define CSL_INTC_MAPPED_NONE (-1)
None mapped

10.6 Typedefs

typedef void(* CSL_IntcEventHandler) (void *)
Event Handler pointer.Event handlers ought to conform to this type

typedef Uint32 CSL_IntcGlobalEnableState
Global Interrupt enable state

typedef CSL_IntcVectId CSL_IntcParam
INTC module parameters for open.This is equivalent to the Vector Id for the event number

typedef Int CSL_IntcEventId
Interrupt Event IDs

typedef struct CSL_IntcObj* CSL_IntcHandle
The interrupt handle. This is returned by the CSL_intcOpen(..) API. The handle is used to identify the event of interest in all INTC calls.

typedef Uint32 CSL_IntcEventEnableState
Event enable state

Chapter 11
MCBSP Module

Topics

11.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within MCBSP module.

This multiple high-speed multichannel buffered serial ports (McBSPs) that allow direct interface to codecs and other devices in a system. The McBSP consists of a data path and a control path that connect to external devices. Separate pins for transmission and reception communicate data to these external devices. Four other pins communicate control information (clocking and frame synchronization). The device communicates to the McBSP using 32-bit-wide control registers accessible via the internal peripheral bus.

Data is communicated to devices interfacing to the McBSP via the data transmit (DX) pin for transmission and via the data receive (DR) pin for reception. Control information (clocking and frame synchronization) is communicated via CLKS, CLKX, CLKR, FSX, and FSR.

The following are McBSP features supported:

- Full-Duplex communication
- Double buffered data registers which allow a continuous data stream.
- Independent framing and clocking for receive and transmit.
- External shift clock generation or an internal programmable frequency shift clock.
- Autobuffering capability through DMA controller
- A wide selection of data sizes2 including 8–, 12–, 16–, 20-, 24-, or 32-bits
- 8-bit data transfers with LSB or MSB first
- Programmable polarity for both frame synchronization and data clocks
- Highly programmable internal clock and frame generation
- Support A-bis mode in normal/32/128 mode (R/XEMODE=0)
- Direct interface to industry standard Codecs, Analog Interface Chips (AICs), and other serially connected A/D and D/A devices.
- Supporting fractional T1/E1. Direct interface to:
- T1/E1 framers
- MVIP switching compatible and ST-BUS compliant devices including:
- MVIP framers
- H.100 framers
- SCSA framers
- IOM-2 compliant devices
- AC97 compliant devices. (The necessary multi-phase frame synchronization provided.)
- IIS compliant devices
- SPI devices

11.2 Functions

This section lists the functions available in the MCBSP module.

11.2.1 CSL_mcbspInit

CSL_Status CSL_mcbspInit (CSL_McbspContext * *pContext*)

Description

This is the initialization function for the McBSP CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use

Arguments

pContext	Pointer to module-context. As McBSP doesn't have any context based information user is expected to pass NULL.

Return Value

CSL_Status

- CSL_SOK - Always returns

Pre Condition

None

Post Condition

The CSL for McBSP is initialized

Modifies

None

Example

```
CSL_Status status;
...
status = CSL_mcbspInit(NULL);
...
```

11.2.2 CSL_mcbspOpen

CSL_McbspHandle CSL_mcbspOpen (CSL_McbspObj * *pMcbspObj*, CSL_InstNum *mcbspNum*, CSL_McbspParam * *pMcbspParam*, CSL_Status * *pStatus*)

Description

This function opens the McBSP CSL. It returns a handle to the McBSP instance. This handle is passed to all other CSL APIs, as the reference to the McBSP instance.The device can be re-opened anytime after it has been normally closed, if so required.

Arguments

```
pMcbspObj        Mcbsp Module Object pointer

mcbspNum         Instance of Mcbsp to be opened

pMcbspParam      Parameter for McBSP

pStatus          Status of the function call
```

Return Value
`CSL_McbspHandle`

- Valid Mcbsp handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The McBSP must be successfully initialized via CSL_mcbspInit() before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid McBSP handle is returned.
- `CSL_ESYS_FAIL` - The McBSP instance is invalid.
- `CSL_ESYS_INVPARAMS` – The Obj structure passed is invalid

2. Mcbsp object structure is populated.

Modifies
1. The status variable
2. Mcbsp object structure

Example

```
CSL_McbspHandle     hMcbsp;
CSL_McbspObj        mcbspObj;
CSL_Status          status;

//Initialize the McBSP CSL
...
hMcbsp = CSL_mcbspOpen(&mcbspObj, CSL_MCBSP_0, NULL, &status);
...
```

11.2.3 CSL_mcbspClose

CSL_Status CSL_mcbspClose (CSL_McbspHandle *hMcbsp*)

Description
Unreserves the McBSP identified by the handle passed. This is a module level close required to invalidate the module handle. The module handle must not be used after this API call.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'
```

Return Value
`CSL_Status`

- `CSL_SOK` – McBSP closed successfully
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspClose().

Post Condition
The mcbsp CSL APIs can not be called until the mcbsp CSL is reopened again using CSL_mcbspOpen().

Modifies
CSL_mcbspObj structure instance values

Example

```
CSL_McbspHandle  hMcbsp;
CSL_McbspObj     mcbspObj;
CSL_Status       status;
...
hMcbsp = CSL_mcbspOpen (&mcbspObj,CSL_MCBSP_0, NULL, &status);
...
status = CSL_mcbspClose(hMcbsp);
...
```

11.2.4 CSL_mcbspHwSetup

CSL_Status CSL_mcbspHwSetup	**(** [CSL_McbspHandle](#)	***hMcbsp*,**
	[CSL_McbspHwSetup](#) *****	***setup***
	)	

Description
This function initializes the device registers with the appropriate values provided through the HwSetup Data structure. After the Setup is completed, the device is ready for operation. For information passed through the HwSetup Data structure, refer CSL_McbspHwSetup.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'

setup           Pointer to setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hwsetup is successfully completed

- `CSL_ESYS_INVPARAMS` - The param passed is invalid
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspHwSetup().

Post Condition
Mcbsp registers are configured according to the hardware setup parameters.

Modifies
MCBSP registers

Example

```
CSL_McbspHandle      hMcbsp;
CSL_McbspHwSetup     hwSetup;
CSL_McbspDataSetup   rxDataSetup = CSL_MCBSP_DATASETUP_DEFAULTS;
CSL_McbspDataSetup   txDataSetup = CSL_MCBSP_DATASETUP_DEFAULTS;
CSL_McbspClkSetup    clkSetup = CSL_MCBSP_CLOCKSETUP_DEFAULTS;
CSL_McbspGlobalSetup glbSetup = CSL_MCBSP_GLOBALSETUP_DEFAULTS;
CSL_McbspMulChSetup  mulChSetup = CSL_MCBSP_MULTICHAN_DEFAULTS;
CSL_Status           status;
...

// Init Successfully done
...
// Open Successfully done
...
hwSetup.global= &glbSetup;
hwSetup.rxdataset = &rxDataSetup;
hwSetup.txdataset= &txDataSetup;
hwSetup.clkset =  &clkSetup;
hwSetup.mulCh =  &mulChSetup;
hwSetup.emumode= CSL_MCBSP_EMU_FREERUN;
hwSetup.extendSetup=NULL;

status = CSL_mcbspHwSetup(hMcbsp, &hwSetup);
...
```

11.2.5 CSL_mcbspHwSetupRaw

CSL_Status CSL_mcbspHwSetupRaw **(** CSL_McbspHandle ***hMcbsp*,**
CSL_McbspConfig ***** ***config***
)

Description
This function initializes the device registers with the register-values provided through the Config Data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

```
hMcbsp        Handle to the Mcbsp instance

config        Pointer to config structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration structure is not properly initialized

Pre Condition
Both CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling this function.

Post Condition
The registers of the specified MCBSP instance will be configured according to value passed.

Modifies
Hardware registers of the specified MCBSP instance.

Example

```
CSL_McbspHandle     hMcbsp;
CSL_McbspConfig     config = CSL_MCBSP_CONFIG_DEFAULTS;
CSL_Status          status;
...
status = CSL_mcbspHwSetupRaw(hMcbsp, &config);
...
```

11.2.6 CSL_mcbspRead

CSL_Status CSL_mcbspRead	**(**	[CSL_McbspHandle]()	***hMcbsp*,**
		[CSL_McbspWordLen]()	***wordLen*,**
		void *	***data***
	)		

Description
This function reads the data from MCBSP. The word length for the read operation is specified using *wordLen* argument. According to this word length, appropriate amount of data will be read in the data object (variable); the pointer to which is passed as the third argument.

Arguments

```
hMcbsp        Handle to the Mcbsp instance

wordLen       Word length of data to be read in

data          Pointer to data object (variable) that will hold
              the read data
```

Return Value
`CSL_Status`

- `CSL_SOK` – Read data successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_EMCBSP_INVSIZE` - Invalid Word length
- `CSL_ESYS_INVPARAMS` – Invalid data pointer

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspRead().

Post Condition
None

Modifies
MCBSP registers

Example:

```
Uint16     inData;
CSL_Status status;
CSL_McbspHandle hMcbsp;
// define MCBSP object, HwSetup structure & initialize McBSP
// Init, Open, HwSetup successfully done in that order
...
// MCBSP SRG, Frame sync, RCV taken out of reset in that order
...
status = CSL_mcbspRead(hMcbsp,
                       CSL_MCBSP_WORDLEN_16,&inData);
...
```

11.2.7 CSL_mcbspWrite

CSL_Status CSL_mcbspWrite	**(**	CSL_McbspHandle	***hMcbsp*, **
		CSL_McbspWordLen	***wordLen*, **
		void *	***data***
	)		

Description
This function transmits the data from MCBSP. The word length for the write operation is specified using *wordLen* argument. According to this word length, the appropriate amount of data will be transmitted from the data object (variable); the pointer to which is passed as the third argument.

Arguments

```
hMcbsp        Handle to the Mcbsp instance

wordLen       Word length of data to be transmitted

data          Pointer to data object (variable) that holds the
```

```
data to be sent out
```

Return Value
`CSL_Status`

- `CSL_SOK` - Write data successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_EMCBSP_INVSIZE` - Invalid word length
- `CSL_ESYS_INVPARAMS` – Invalid data pointer

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_ *mcbspWrite*().

Post Condition
Data is written to DXR register

Modifies
McBSP registers

Example:
```
Uint16     outData;
CSL_Status status;
CSL_McbspHandle hMcbsp;
...
// MCBSP object defined and HwSetup structure defined and
initialized
// Init, Open, HwSetup successfully done in that order
...
// MCBSP SRG, Frame sync, XMT taken out of reset in that order
...
outData = 0x1234;
status = CSL_mcbspWrite(hMcbsp,
                        CSL_MCBSP_WORDLEN_16,&outData);
...
```

11.2.8 CSL_mcbspIoWrite

void CSL_mcbspIoWrite	**(**	[CSL_McbspHandle](#)	***hMcbsp***,
		CSL_BitMask16	***outputSel***,
		Uint16	***outputData***
	)		

Description
This function sends the data using MCBSP pin, which is configured as general purpose output. The 16-bit data transmitted is specified by 'outputData' argument. MCBSP pin to use in this write operation is identified by the second argument.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'
```

```
outputSel       MCBSP pin to be used as general purpose output

outputData      1 bit output data to be transmitted
```

Return Value
None

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspIoWrite().

Post Condition
None

Modifies
McBSP registers

Example

```
Uint16              outData;
CSL_Status          status;
CSL_McbspHandle     hMcbsp;
...
// MCBSP object defined and HwSetup structure defined and
// initialized
...
// Init, Open, HwSetup successfully done in that order
...
outData = 1;
CSL_mcbspIoWrite(hMcbsp, CSL_MCBSP_IO_CLKX, outData);
 ...
```

11.2.9 CSL_mcbspIoRead

Uint16 CSL_mcbspIoRead	**(**	CSL_McbspHandle	***hMcbsp*,**
		CSL_BitMask16	***inputSel***
	)		

Description
This function reads the data from MCBSP pin, which is configured as general purpose input. The 16-bit data read from this pin is returned by this API. MCBSP pin to use in this read operation is identified by the second argument.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'

inputSel        MCBSP pin to be used as general purpose input
```

Return Value
`Uint16`

- Data read from the pin

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspIoRead().

Post Condition
None

Modifies
None

Example

```
Uint16              inData;
Uint16              clkx_data;
Uint16              clkr_data;
CSL_Status          status;
CSL_BitMask16       inMask;
CSL_McbspHandle     hMcbsp;

// MCBSP object defined and HwSetup structure defined and
// initialized

// Init, Open, HwSetup successfully done in that order
...
inMask = CSL_MCBSP_IO_CLKX | CSL_MCBSP_IO_CLKR;
inData = CSL_mcbspIoRead(hMcbsp, inMask);
if ((inData & CSL_MCBSP_IO_CLKX) != 0)
    clkx_data = 1;
else
    clkx_data = 0;
if ((inData & CSL_MCBSP_IO_CLKR) != 0)
    clkr_data = 1;
else
    clkr_data = 0;
...
```

11.2.10 CSL_mcbspHwControl

CSL_Status CSL_mcbspHwControl	**(**	CSL_McbspHandle	***hMcbsp*,**
		CSL_McbspControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of MCBSP.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'
```

```
cmd             Control command

arg             Optional argument as per the control command
```

Return Value
CSL_Status

- CSL_SOK - Command successful
- CSL_ESYS_INVCMD - The Command passed is invalid
- CSL_ESYS_BADHANDLE - The handle passed is invalid
- CSL_ESYS_INVPARAMS – The parameter is invalid

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspHwControl().

Post Condition
Mcbsp registers are configured according to the command passed.

Modifies
McBSP registers determined by the command

Example

```
CSL_Status          status;
CSL_BitMask16       ctrlMask;
CSL_McbspHandle     hMcbsp;

// MCBSP object defined and HwSetup structure defined and
// initialized

// Init successfully done

// Open successfully done

// HwSetup sucessfully done

// MCBSP SRG and Frame sync taken out of reset
...
ctrlMask = CSL_MCBSP_CTRL_RX_ENABLE | CSL_MCBSP_CTRL_TX_ENABLE;
status = CSL_mcbspHwControl(hMcbsp,
                            CSL_MCBSP_CMD_RESET_CONTROL,
                            &ctrlMask);
...
```

11.2.11 CSL_mcbspGetHwStatus

CSL_Status CSL_mcbspGetHwStatus	**(**	CSL_McbspHandle	***hMcbsp*, **
		CSL_McbspHwStatusQuery	***myQuery*, **
		void *	***response***
	)		

Description
This function gets the status of different operations or some setup-parameters of MCBSP. The status is returned through the third parameter.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'

myQuery         Query command

response        Response from the query. Pointer to appropriate
                object corresponding to the query command needs
                to be passed here
```

Return Value
`CSL_Status`

- `CSL_SOK` - Query successful
- `CSL_ESYS_INVQUERY` - The Query passed is invalid
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` – The parameter is invalid

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspGetHwStatus().

Post Condition
None

Modifies
Third parameter “response” vlaue

Example

```
CSL_McbspHandle     hMcbsp;
CSL_Status          status;
Uint16              response;

// MCBSP object defined and HwSetup structure defined and
// initialized

// Init successfully done

// Open successfully done
...
status = CSL_mcbspGetHwStatus(hMcbsp,
                              CSL_MCBSP_QUERY_DEV_STATUS,
                              &response);
if (response & CSL_MCBSP_RRDY)
{
      // Receiver is ready to with new data
      ...
}
...
```

11.2.12 CSL_mcbspGetHwSetup

CSL_Status CSL_mcbspGetHwSetup **(** CSL_McbspHandle ***hMcbsp***, CSL_McbspHwSetup * ***myHwSetup*** **)**

Description
This function gets the status of some or all of the setup-parameters of MCBSP. To get the status of complete MCBSP h/w setup, all the sub-structure pointers inside the main HwSetup structure, should be non-NULL.

Arguments

```
hMcbsp          MCBSP handle returned by successful 'open'

myHwSetup       Pointer to CSL_McbspHwSetup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Get hwsetup successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` – Invalid setup structure

Pre Condition
CSL_mcbspInit() and CSL_mcbspOpen() must be called successfully in order before calling CSL_mcbspGetHwSetup().

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
None

Example

```
     CSL_McbspHandle      hMcbsp;
     CSL_Status           status;
     CSL_McbspHwSetup     readSetup;
     CSL_McbspGlobalSetup gblSetup;
     CSL_McbspClkSetup    clkSetup;
     CSL_McbspDataSetup   rxDataSetup;
     CSL_McbspDataSetup   txDataSetup;

    // MCBSP object defined and HwSetup structure defined and
           initialized

    // Init successfully done

    // Open successfully done
    ...
    readSetup.global    = &gblSetup;
    readSetup.rxdataset = &rxDataSetup;
```

```
    readSetup.txdataset = &txDataSetup;
    readSetup.clkset    = &clkSetup;
    status = CSL_mcbspGetHwSetup(hMcbsp, &readSetup);
    ...
```

11.2.13 CSL_mcbspGetBaseAddress

CSL_Status CSL_mcbspGetBaseAddress	**(**	**CSL_InstNum**	***mcbspNum*,**
		CSL_McbspParam *	***pMcbspParam*,**
		CSL_McbspBaseAddress *	***pBaseAddress***
	)		

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_mcbspOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

`mcbspNum`	`Specifies the instance of the MCBSP to be opened`
`pMcbspParam`	`Module specific parameters`
`pBaseAddress`	`Pointer to baseaddress structure`

Return Value
`CSL_Status`

- `CSL_SOK` - Successfull on getting the base address of McBSP.
- `CSL_ESYS_FAIL` - The instance number is invalid.
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
None

Post Condition
Base Address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
    CSL_Status              status;
    CSL_McbspBaseAddress    baseAddress;
    ...
    status = CSL_mcbspGetBaseAddress(CSL_MCBSP_0, NULL,
                                     &baseAddress);
    ...
```

11.3 Data Structures

This section lists the data structures available in the MCBSP module.

11.3.1 CSL_McbspObj

Detailed Description

This structure/object holds the context of the instance of MCBSP opened using CSL_mcbspOpen() function. Pointer to this object is passed as MCBSP Handle to all MCBSP CSL APIs. CSL_mcbspOpen() function initializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_McbspObj::perNum

Instance of MCBSP being referred by this object

CSL_McbspRegsOvly CSL_McbspObj::regs

Pointer to the register overlay structure of the MCBSP

11.3.2 CSL_McbspConfig

Detailed Description

This is configuration structure of MCBSP. This is used to configure MCBSP using CSL_HwSetupRaw function. This is a structure of register values, rather than a structure of register field values like CSL_McbspHwSetup.

Field Documentation

volatile Uint32 CSL_McbspConfig::MCR

Multichannel Control Register

volatile Uint32 CSL_McbspConfig::PCR

Pin Control Register

volatile Uint32 CSL_McbspConfig::RCERE0

Receive Channel Enable Register for Partition A and B

volatile Uint32 CSL_McbspConfig::RCERE1

Receive Channel Enable Register for Partition C and D

volatile Uint32 CSL_McbspConfig::RCERE2

Receive Channel Enable Register for Partition E and F

volatile Uint32 CSL_McbspConfig::RCERE3

Receive Channel Enable Register for Partition G and H

volatile Uint32 CSL_McbspConfig::RCR

Receive Control Register

volatile Uint32 CSL_McbspConfig::SPCR

Serial Port Control Register

volatile Uint32 CSL_McbspConfig::SRGR

Sample Rate Generator Register

volatile Uint32 CSL_McbspConfig::XCERE0
Transmit Channel Enable Register for Partition A and B

volatile Uint32 CSL_McbspConfig::XCERE1
Transmit Channel Enable Register for Partition C and D

volatile Uint32 CSL_McbspConfig::XCERE2
Transmit Channel Enable Register for Partition E and F

volatile Uint32 CSL_McbspConfig::XCERE3
Transmit Channel Enable Register for Partition G and H

volatile Uint32 CSL_McbspConfig::XCR
Transmit Control Register

11.3.3 CSL_McbspContext

Detailed Description
This contains McBSP specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_McbspContext::contextInfo
McBSP context information. The declaration is just a placeholder for future implementation.

11.3.4 CSL_McbspHwSetup

Detailed Description
This is the hardware setup structure for configuring MCBSP using CSL_mcbspHwSetup() function.

Field Documentation

CSL_McbspClkSetup* CSL_McbspHwSetup::clkset
Clock configuration parameters

CSL_McbspEmu CSL_McbspHwSetup::emumode
Emulation mode parameters

void* CSL_McbspHwSetup::extendSetup
Any extra parameters, for future use

CSL_McbspGlobalSetup* CSL_McbspHwSetup::global
Global configuration parameters

CSL_McbspMulChSetup* CSL_McbspHwSetup::mulCh
Multichannel mode configuration parameters

CSL_McbspDataSetup* CSL_McbspHwSetup::rxdataset
RCV data setup related parameters

CSL_McbspDataSetup* CSL_McbspHwSetup::txdataset

XMT data setup related parameters

11.3.5 CSL_McbspParam

Detailed Description
This contains the MCBSP specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_McbspParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

11.3.6 CSL_McbspBaseAddress

Detailed Description
This will have the base-address information for the peripheral instance

Field Documentation

CSL_McbspRegsOvly CSL_McbspBaseAddress::regs
Base-address of the Configuration registers of MCBSP

11.3.7 CSL_McbspBlkAssign

Detailed Description
Pointer to this structure is used as the third argument in CSL_mcbspHwControl() for block assignment in multichannel mode

Field Documentation

CSL_McbspBlock CSL_McbspBlkAssign::block
Block to choose

CSL_McbspPartition CSL_McbspBlkAssign::partition
Partition to choose

11.3.8 CSL_McbspChanControl

Detailed Description
Pointer to this structure is used as the third argument in CSL_mcbspHwControl() for channel control operations (Enable/Disable TX/RX) in multichannel mode.

Field Documentation

Uint16 CSL_McbspChanControl::channelNo
Channel number to control

CSL_McbspChCtrl CSL_McbspChanControl::operation
Control operation

11.3.9 CSL_McbspDataSetup

Detailed Description

This is a sub-structure in CSL_McbspHwSetup. This structure is used for configuring input/output data related parameters.

Field Documentation

CSL_McbspCompand CSL_McbspDataSetup::compand
Companding options

CSL_McbspDataDelay CSL_McbspDataSetup::dataDelay
Data delay in number of bits

Uint16 CSL_McbspDataSetup::frmLength1
Number of words per frame in phase 1

Uint16 CSL_McbspDataSetup::frmLength2
Number of words per frame in phase 2

CSL_McbspFrmSync CSL_McbspDataSetup::frmSyncIgn
Frame Sync ignore

CSL_McbspPhase CSL_McbspDataSetup::numPhases
Number of phases in a frame

CSL_McbspRjustDxena CSL_McbspDataSetup::rjust_dxenable
Controls DX delay for XMT or sign-extension and justification for RCV

CSL_McbspWordLen CSL_McbspDataSetup::wordLength1
Number of bits per word in phase 1

CSL_McbspWordLen CSL_McbspDataSetup::wordLength2
Number of bits per word in phase 2

CSL_McbspBitReversal CSL_McbspDataSetup::wordReverse
32-bit reversal feature

CSL_McbspIntMode CSL_McbspDataSetup:: IntEvent
Interrupt event mask

11.3.10 CSL_McbspClkSetup

Detailed Description
This is a sub-structure in *CSL_McbspHwSetup*. This structure is used for configuring Clock and Frame Sync generation parameters.

Field Documentation

CSL_McbspTxRxClkMode CSL_McbspClkSetup::clkRxMode
RCV clock mode

CSL_McbspClkPol CSL_McbspClkSetup::clkRxPolarity
RCV clock polarity

CSL_McbspTxRxClkMode CSL_McbspClkSetup::clkTxMode
XMT clock mode

CSL_McbspClkPol CSL_McbspClkSetup::clkTxPolarity
XMT clock polarity

CSL_McbspFsClkMode CSL_McbspClkSetup::frmSyncRxMode
RCV frame sync mode

CSL_McbspFsPol CSL_McbspClkSetup::frmSyncRxPolarity
RCV frame sync polarity

CSL_McbspFsClkMode CSL_McbspClkSetup::frmSyncTxMode
XMT frame sync mode

CSL_McbspFsPol CSL_McbspClkSetup::frmSyncTxPolarity
XMT frame sync polarity

Uint16 CSL_McbspClkSetup::srgClkDivide
SRG divide-down ratio

CSL_McbspClkPol CSL_McbspClkSetup::srgClkPolarity
SRG clock polarity

CSL_McbspClkgSyncMode CSL_McbspClkSetup::srgClkSync
SRG clock synchronization mode

Uint16 CSL_McbspClkSetup::srgFrmPeriod
SRG frame sync period

Uint16 CSL_McbspClkSetup::srgFrmPulseWidth
SRG frame sync pulse width

CSL_McbspSrgClk CSL_McbspClkSetup::srgInputClkMode
SRG input clock mode

CSL_McbspTxFsMode CSL_McbspClkSetup::srgTxFrmSyncMode
SRG XMT frame-synchronization mode

11.3.11 CSL_McbspGlobalSetup

Detailed Description
This is a sub-structure in *CSL_McbspHwSetup*. This structure is used for configuring the parameters global to MCBSP

Field Documentation

CSL_McbspClkStp CSL_McbspGlobalSetup::clkStopMode
Clock stop mode

CSL_McbspDlbMode CSL_McbspGlobalSetup::dlbMode
Digital Loopback mode

CSL_McbspIOMode CSL_McbspGlobalSetup::ioEnableMode
XMT and RCV IO enable bit

11.3.12 CSL_McbspMulChSetup

Detailed Description

This is a sub-structure in *CSL_McbspHwSetup*. This structure is used for configuring Multichannel mode parameters

Field Documentation

Uint16 CSL_McbspMulChSetup::rxMulChSel
RCV multichannel selection mode

CSL_McbspPABlk CSL_McbspMulChSetup::rxPartABlk
RCV partition A block

CSL_McbspPBBlk CSL_McbspMulChSetup::rxPartBBlk
RCV partition B block

CSL_McbspPartMode CSL_McbspMulChSetup::rxPartition
RCV partition

Uint16 CSL_McbspMulChSetup::txMulChSel
XMT multichannel selection mode

CSL_McbspPABlk CSL_McbspMulChSetup::txPartABlk
XMT partition A block

CSL_McbspPBBlk CSL_McbspMulChSetup::txPartBBlk
XMT partition B block

CSL_McbspPartMode CSL_McbspMulChSetup::txPartition
XMT partition

11.4 Enumerations

11.4.1 CSL_McbspWordLen

enum CSL_McbspWordLen
This enumeration contains the word lengths supported on MCBSP. Use this symbol for setting Word Length in each Phase for every Frame.

Enumeration values:

CSL_MCBSP_WORDLEN_8	Word Length for Frame is 8
CSL_MCBSP_WORDLEN_12	Word Length for Frame is 12
CSL_MCBSP_WORDLEN_16	Word Length for Frame is 16
CSL_MCBSP_WORDLEN_20	Word Length for Frame is 20
CSL_MCBSP_WORDLEN_24	Word Length for Frame is 24
CSL_MCBSP_WORDLEN_32	Word Length for Frame is 32

11.4.2 CSL_McbspCompand

enum CSL_McbspCompand
MCBSP companding options - Use this symbol to set Companding related options.

Enumeration values:

CSL_MCBSP_COMPAND_OFF_MSB_FIRST	No companding for msb
CSL_MCBSP_COMPAND_OFF_LSB_FIRST	No companding for lsb
CSL_MCBSP_COMPAND_MULAW	mu-law comapanding enable for channel
CSL_MCBSP_COMPAND_ALAW	A-law comapanding enable for channel

11.4.3 CSL_McbspDataDelay

enum CSL_McbspDataDelay
Data delay in bits - Use this symbol to set XMT/RCV Data Delay (in bits).

Enumeration values:

CSL_MCBSP_DATADELAY_0_BIT	Sets XMT/RCV Data Delay is 0
CSL_MCBSP_DATADELAY_1_BIT	Sets XMT/RCV Data Delay is 1
CSL_MCBSP_DATADELAY_2_BITS	Sets XMT/RCV Data Delay is 2

11.4.4 CSL_McbspIntMode

enum CSL_McbspIntMode
This enumeration contains McBSP Interrupts modes - Use this symbol to set Interrupt mode (i.e. source of interrupt generation). This symbol is used on both RCV and XMT for RINT and XINT generation mode.

Enumeration values:

CSL_MCBSP_INTMODE_ON_READY	Interrupt generated on RRDY of RCV or XRDY of XMT
CSL_MCBSP_INTMODE_ON_EOB	Interrupt generated on end of 16-channel block transfer in multichannel mode
CSL_MCBSP_INTMODE_ON_FSYNC	Interrupt generated on frame sync
CSL_MCBSP_INTMODE_ON_SYNCERR	Interrupt generated on synchronization error

11.4.5 CSL_McbspFsClkMode

enum CSL_McbspFsClkMode
Frame sync clock source - Use this symbol to set the frame sync clock source as internal or external.

Enumeration values:

CSL_MCBSP_FSCLKMODE_EXTERNAL	Frame sync clock source as external
CSL_MCBSP_FSCLKMODE_INTERNAL	Frame sync clock source as internal

11.4.6 CSL_McbspTxRxClkMode

enum CSL_McbspTxRxClkMode
Clock source - Use this symbol to set the clock source as internal or external.

Enumeration values:

CSL_MCBSP_TXRXCLKMODE_EXTERNAL	Clock source as external
CSL_MCBSP_TXRXCLKMODE_INTERNAL	Clock source as internal

11.4.7 CSL_McbspFsPol

enum CSL_McbspFsPol
Frame sync polarity - Use this symbol to set frame sync polarity as active-high or active-low.

Enumeration values:

CSL_MCBSP_FSPOL_ACTIVE_HIGH	Frame sync polarity is active-high
CSL_MCBSP_FSPOL_ACTIVE_LOW	Frame sync polarity is active-low

11.4.8 CSL_McbspClkPol

enum CSL_McbspClkPol
Clock polarity - Use this symbol to set XMT or RCV clock polarity as rising or falling edge.

Enumeration values:

CSL_MCBSP_CLKPOL_TX_RISING_EDGE	XMT clock polarity is rising edge
CSL_MCBSP_CLKPOL_RX_FALLING_EDGE	RCV clock polarity is falling edge
CSL_MCBSP_CLKPOL_SRG_RISING_EDGE	SRG clock polarity is rising edge
CSL_MCBSP_CLKPOL_TX_FALLING_EDGE	XMT clock polarity is falling edge
CSL_MCBSP_CLKPOL_RX_RISING_EDGE	RCV clock polarity is rising edge

CSL_MCBSP_CLKPOL_SRG_FALLING_EDGE	SRG clock polarity Is falling edge

11.4.9 CSL_McbspSrgClk

enum CSL_McbspSrgClk
SRG clock source - Use this symbol to select input clock source for Sample Rate Generator.

Enumeration values:

CSL_MCBSP_SRGCLK_CLKS	Input clock source for Sample Rate Generator is CLKS pin
CSL_MCBSP_SRGCLK_CLKCPU	Input clock source for Sample Rate Generator is CPU

11.4.10 CSL_McbspTxFsMode

enum CSL_McbspTxFsMode
This enumeration contains XMT Frame Sync generation mode - Use this symbol to set XMT Frame Sync generation mode.

Enumeration values:

CSL_MCBSP_TXFSMODE_DXRCOPY	Disables the frame sync generation mode
CSL_MCBSP_TXFSMODE_SRG	Enables the frame sync generation mode

11.4.11 CSL_McbspIOMode

enum CSL_McbspIOMode
XMT and RCV IO Mode - Use this symbol to Enable/Disable IO Mode for XMT and RCV.

Enumeration values:

CSL_MCBSP_IOMODE_TXDIS_RXDIS	Disable the both XMT and RCV IO mode
CSL_MCBSP_IOMODE_TXDIS_RXEN	Disable XMT and enable RCV IO mode
CSL_MCBSP_IOMODE_TXEN_RXDIS	Enable XMT and Disable RCV IO mode
CSL_MCBSP_IOMODE_TXEN_RXEN	Enable XMT and enable RCV IO mode

11.4.12 CSL_McbspClkStp

enum CSL_McbspClkStp
Clock Stop Mode - Use this symbol to Enable/Disable Clock Stop Mode.

Enumeration values:

CSL_MCBSP_CLKSTP_DISABLE	Disable the clock stop mode
CSL_MCBSP_CLKSTP_WITHOUT_DELAY	Enable the clock stop mode with out delay
CSL_MCBSP_CLKSTP_WITH_DELAY	Enable the clock stop mode with delay

11.4.13 CSL_McbspPartMode

enum CSL_McbspPartMode
This enumeration contains the multichannel mode partition type - Use this symbol to select the partition type in multichannel mode.

Enumeration values:

CSL_MCBSP_PARTMODE_2PARTITION	Two partition mode
CSL_MCBSP_PARTMODE_8PARTITION	Eight partition multichannel mode

11.4.14 CSL_McbspPABlk

enum CSL_McbspPABlk
Multichannel mode PartitionA block - Use this symbol to assign Blocks to Partition-A in multichannel mode

Enumeration values:

CSL_MCBSP_PABLK_0	Block 0 for partition A
CSL_MCBSP_PABLK_2	Block 2 for partition A
CSL_MCBSP_PABLK_4	Block 4 for partition A
CSL_MCBSP_PABLK_6	Block 6 for partition A

11.4.15 CSL_McbspPBBlk

enum CSL_McbspPBBlk
Multichannel mode PartitionB block - Use this symbol to assign Blocks to Partition-B in multichannel mode

Enumeration values:

CSL_MCBSP_PBBLK_1	Block 1 for partition B
CSL_MCBSP_PBBLK_3	Block 3 for partition B
CSL_MCBSP_PBBLK_5	Block 5 for partition B
CSL_MCBSP_PBBLK_7	Block 7 for partition B

11.4.16 CSL_McbspEmu

enum CSL_McbspEmu
Emulation mode setting - Use this symbol to set the Emulation Mode

Enumeration values:

CSL_MCBSP_EMU_STOP	Emulation mode stop
CSL_MCBSP_EMU_TX_STOP	Emulation mode TX stop
CSL_MCBSP_EMU_FREERUN	Emulation free run mode

11.4.17 CSL_McbspPartition

enum CSL_McbspPartition
Multichannel mode Partition select - Use this symbol in multichannel mode to select the Partition for assigning a block to

Enumeration values:

CSL_MCBSP_PARTITION_ATX	TX partition for A
CSL_MCBSP_PARTITION_ARX	RX partition for A
CSL_MCBSP_PARTITION_BTX	TX partition for B
CSL_MCBSP_PARTITION_BRX	RX partition for B

11.4.18 CSL_McbspBlock

enum CSL_McbspBlock
Multichannel mode Block select - Use this symbol in multichannel mode to select block on which the operation is to be performed

Enumeration values:

CSL_MCBSP_BLOCK_0	Block 0 for multichannel mode
CSL_MCBSP_BLOCK_1	Block 1 for multichannel mode
CSL_MCBSP_BLOCK_2	Block 2 for multichannel mode
CSL_MCBSP_BLOCK_3	Block 3 for multichannel mode
CSL_MCBSP_BLOCK_4	Block 4 for multichannel mode
CSL_MCBSP_BLOCK_5	Block 5 for multichannel mode
CSL_MCBSP_BLOCK_6	Block 6 for multichannel mode
CSL_MCBSP_BLOCK_7	Block 7 for multichannel mode

11.4.19 CSL_McbspChCtrl

enum CSL_McbspChCtrl
Channel control in multichannel mode Use this symbol to enable/disable a channel in multichannel mode. This is a member of CSL_McbspChanControl structure, which is input to CSL_mcbspHwControl() function for CSL_MCBSP_CMD_CHANNEL_CONTROL command.

Enumeration values:

CSL_MCBSP_CHCTRL_TX_ENABLE	TX enable for multichannel mode
CSL_MCBSP_CHCTRL_TX_DISABLE	TX disable for multichannel mode
CSL_MCBSP_CHCTRL_RX_ENABLE	RX enable for multichannel mode
CSL_MCBSP_CHCTRL_RX_DISABLE	RX disable for multichannel mode

11.4.20 CSL_McbspChType

enum CSL_McbspChType
Channel type: TX, RX or both - Use this symbol to select the channel type for CSL_mcbspHwControl()

Enumeration values:

CSL_MCBSP_CHTYPE_RX	Channel type is RX
CSL_MCBSP_CHTYPE_TX	Channel type is TX
CSL_MCBSP_CHTYPE_TXRX	Channel type is TXRX

11.4.21 CSL_McbspDlbMode

enum CSL_McbspDlbMode
Digital Loopback mode selection - Use this symbol to enable/disable digital loopback mode

Enumeration values:

CSL_MCBSP_DLBMODE_OFF	Disable digital loopback mode
CSL_MCBSP_DLBMODE_ON	Enable digital loopback mode

11.4.22 CSL_McbspPhase

enum CSL_McbspPhase
Phase count selection - Use this symbol to select number of phases per frame

Enumeration values:

CSL_MCBSP_PHASE_SINGLE	Single phase for frame
CSL_MCBSP_PHASE_DUAL	Dual phase for frame

11.4.23 CSL_McbspFrmSync

enum CSL_McbspFrmSync
Frame sync ignore status - Use this symbol to detect or ignore frame synchronization

Enumeration values:

CSL_MCBSP_FRMSYNC_DETECT	Detect frame synchronization
CSL_MCBSP_FRMSYNC_IGNORE	Ignore frame synchronization

11.4.24 CSL_McbspRjustDxena

enum CSL_McbspRjustDxena
RJUST or DXENA settings - Use this symbol for setting up RCV sign-extension and justification mode or enabling/disabling XMT DX pin delay

Enumeration values:

CSL_MCBSP_RJUSTDXENA_RJUST_RZF	RCV setting - right justify, fill MSBs with zeros
CSL_MCBSP_RJUSTDXENA_DXENA_OFF	XMT setting - Delay at DX pin disabled
CSL_MCBSP_RJUSTDXENA_RJUST_RSE	RCV setting - right justify, sign-extend the data into MSBs
CSL_MCBSP_RJUSTDXENA_DXENA_ON	XMT setting - Delay at DX pin enabled
CSL_MCBSP_RJUSTDXENA_RJUST_LZF	RCV setting - left justify, fill LSBs with zeros

11.4.25 CSL_McbspClkgSyncMode

enum CSL_McbspClkgSyncMode
CLKG sync mode selection - Use this symbol to enable/disable CLKG synchronization when input CLK source for SRGR is external

Enumeration values:

CSL_MCBSP_CLKGSYNCMODE_OFF	Disable CLKG synchronization
CSL_MCBSP_CLKGSYNCMODE_ON	Enable CLKG synchronization

11.4.26 CSL_McbspRstStat

enum CSL_McbspRstStat
Tx/Rx reset status - Use this symbol to compare the output of CSL_mcbspGetHwStatus() for CSL_MCBSP_QUERY_TX_RST_STAT and CSL_MCBSP_QUERY_RX_RST_STAT queries

Enumeration values:

CSL_MCBSP_RSTSTAT_TX_IN_RESET	Disable the XRST bit
CSL_MCBSP_RSTSTAT_RX_IN_RESET	Disable the RRST bit
CSL_MCBSP_RSTSTAT_TX_OUTOF_RESET	Enable the XRST bit
CSL_MCBSP_RSTSTAT_RX_OUTOF_RESET	Enable the RRST bit

11.4.27 CSL_McbspBitReversal

enum CSL_McbspBitReversal
McBSP 32-bit reversal feature

Enumeration values:

CSL_MCBSP_32BIT_REVERS_DISABLE	32-bit reversal disabled
CSL_MCBSP_32BIT_REVERS_ENABLE	32-bit reversal enabled. 32-bit data is received LSB first. Word length should be set for 32-bit operation; else operation undefined

11.4.28 CSL_McbspControlCmd

enum CSL_McbspControlCmd
This is the set of control commands that are passed to CSL_mcbspHwControl(), with an optional argument type-casted to void* The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_MCBSP_CMD_ASSIGN_BLOCK	Assigns a block to a particular partition in multichannel mode. **Parameters:** *(CSL_McbspBlkAssign* *)
CSL_MCBSP_CMD_CHANNEL_CONTROL	Enables or disables a channel in multichannel mode. **Parameters:** *(CSL_McbspChanControl* *)
CSL_MCBSP_CMD_CLEAR_FRAME_SYNC	Clears frame sync error for XMT or RCV. **Parameters:** *(CSL_McbspChType* *)
CSL_MCBSP_CMD_RESET	Resets all the registers to their power-on default values. **Parameters:** *None*
CSL_MCBSP_CMD_RESET_CONTROL	Enable/Disable - Frame Sync, Sample Rate Generator and XMT/RCV Operation. **Parameters:** *(CSL_BitMask16* *)

11.4.29 CSL_McbspHwStatusQuery

enum CSL_McbspHwStatusQuery
This is the set of query commands to get the status of various operations in McBSP. The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_MCBSP_QUERY_CUR_TX_BLK	Queries the current XMT block. **Parameters:** *(CSL_McbspBlock* *)
CSL_MCBSP_QUERY_CUR_RX_BLK	Queries the current RCV block. **Parameters:** *(CSL_McbspBlock* *)

CSL_MCBSP_QUERY_DEV_STATUS	Queries the status of RRDY, XRDY, RFULL, XEMPTY, RSYNCERR and XSYNCERR events and returns them in supplied CSL_BitMask16 argument. **Parameters:** *(CSL_BitMask16 *)*
CSL_MCBSP_QUERY_TX_RST_STAT	Queries XMT reset status. **Parameters:** *(CSL_McbspRstStat *)* **Returns:** CSL_SOK
CSL_MCBSP_QUERY_RX_RST_STAT	Queries RCV reset status. **Parameters:** *(CSL_McbspRstStat *)*

11.5 Macros

#define CSL_EMCBSP_INVCNTLCMD (CSL_EMCBSP_FIRST - 0)
Invalid Control Command

#define CSL_EMCBSP_INVMODE (CSL_EMCBSP_FIRST - 5)
Invalid mode to conduct operation

#define CSL_EMCBSP_INVPARAMS (CSL_EMCBSP_FIRST - 2)
Invalid Parameter

#define CSL_EMCBSP_INVQUERY (CSL_EMCBSP_FIRST - 1)
Invalid Query

#define CSL_EMCBSP_INVSIZE (CSL_EMCBSP_FIRST - 3)
Invalid Size

#define CSL_EMCBSP_NOTEXIST (CSL_EMCBSP_FIRST - 4)
'Does not exist'

**#define CSL_MCBSP_CLOCKSETUP_DEFAULTS **
```
{                 \
 (CSL_McbspFsClkMode)CSL_MCBSP_FSCLKMODE_EXTERNAL,       \
 (CSL_McbspFsClkMode)CSL_MCBSP_FSCLKMODE_EXTERNAL,       \
 (CSL_McbspTxRxClkMode)CSL_MCBSP_TXRXCLKMODE_INTERNAL,   \
 (CSL_McbspTxRxClkMode)CSL_MCBSP_TXRXCLKMODE_EXTERNAL,   \
 (CSL_McbspFsPol)0,                                      \
 (CSL_McbspFsPol)0,                                      \
 (CSL_McbspClkPol)0,                                     \
 (CSL_McbspClkPol)0,                                     \
 1,                                                      \
 0x40,                                                   \
 0xFF,                                                   \
 (CSL_McbspSrgClk)0,                                     \
 (CSL_McbspClkPol)0,                                     \
 (CSL_McbspTxFsMode)CSL_MCBSP_TXFSMODE_SRG,              \
 (CSL_McbspClkgSyncMode)CSL_MCBSP_CLKGSYNCMODE_OFF      }
```
Clock setup defaults

**#define CSL_MCBSP_CONFIG_DEFAULTS **
```
{ \
    CSL_MCBSP_SPCR_RESETVAL,    \
    CSL_MCBSP_RCR_RESETVAL,     \
    CSL_MCBSP_XCR_RESETVAL,     \
    CSL_MCBSP_SRGR_RESETVAL,    \
    CSL_MCBSP_MCR_RESETVAL,     \
    CSL_MCBSP_RCERE0_RESETVAL,  \
    CSL_MCBSP_XCERE0_RESETVAL,  \
    CSL_MCBSP_PCR_RESETVAL,     \
    CSL_MCBSP_RCERE1_RESETVAL,  \
    CSL_MCBSP_XCERE1_RESETVAL,  \
    CSL_MCBSP_RCERE2_RESETVAL,  \
    CSL_MCBSP_XCERE2_RESETVAL,  \
    CSL_MCBSP_RCERE3_RESETVAL,  \
```

```
    CSL_MCBSP_XCERE3_RESETVAL    \
}
```

Default values for Config structure

#define CSL_MCBSP_CTRL_FSYNC_DISABLE (64)
To disable Frame Sync Generation in resetControl Function

#define CSL_MCBSP_CTRL_FSYNC_ENABLE (16)
To enable Frame Sync Generation in resetControl Function

#define CSL_MCBSP_CTRL_RX_DISABLE (4)
To disable Receiver in resetControl Function

#define CSL_MCBSP_CTRL_RX_ENABLE (1)
To enable Receiver in resetControl Function

#define CSL_MCBSP_CTRL_SRG_DISABLE (128)
To disable Sample Rate Generator in resetControl Function

#define CSL_MCBSP_CTRL_SRG_ENABLE (32)
To enable Sample Rate Generator in resetControl Function

#define CSL_MCBSP_CTRL_TX_DISABLE (8)
To disable Transmitter in resetControl Function

#define CSL_MCBSP_CTRL_TX_ENABLE (2)
To enable Transmitter in resetControl Function

**#define CSL_MCBSP_DATASETUP_DEFAULTS **

```
{            \
   (CSL_McbspPhase)CSL_MCBSP_PHASE_SINGLE,          \
   (CSL_McbspWordLen)CSL_MCBSP_WORDLEN_16,          \
   1,                                               \
   (CSL_McbspWordLen)0,                             \
   0,                                               \
   (CSL_McbspFrmSync)CSL_MCBSP_FRMSYNC_DETECT,      \
   (CSL_McbspCompand)CSL_MCBSP_COMPAND_OFF_MSB_FIRST, \
   (CSL_McbspDataDelay)CSL_MCBSP_DATADELAY_0_BIT,   \
   (CSL_McbspRjustDxena)0,                          \
   (CSL_McbspIntMode)CSL_MCBSP_INTMODE_ON_READY,    \
   (CSL_McbspBitReversal)CSL_MCBSP_32BIT_REVERS_DISABLE }
```

Data Setup defaults

#define CSL_MCBSP_EMUMODE_DEFAULT CSL_MCBSP_EMU_STOP
Default Emulation mode - Stop

#define CSL_MCBSP_EXTENDSETUP_DEFAULT NULL
Extend Setup default - NULL

**#define CSL_MCBSP_GLOBALSETUP_DEFAULTS **

```
{               \
 (CSL_McbspIOMode)CSL_MCBSP_IOMODE_TXDIS_RXDIS,         \
 (CSL_McbspDlbMode)CSL_MCBSP_DLBMODE_OFF,               \
 (CSL_McbspClkStp)CSL_MCBSP_CLKSTP_DISABLE }
```

Global parameters Setup defaults

#define CSL_MCBSP_IO_CLKR (8)
I/O Pin Input/Output configuration for CLKR Pin

#define CSL_MCBSP_IO_CLKS (64)
Not Configurable. Always Input.

#define CSL_MCBSP_IO_CLKX (1)
I/O Pin Input/Output configuration for CLKX Pin

#define CSL_MCBSP_IO_DR (32)
Not Configurable. Always Input.

#define CSL_MCBSP_IO_DX (4)
Not Configurable. Always Output.

#define CSL_MCBSP_IO_FSR (16)
I/O Pin Input/Output configuration for FSR Pin

#define CSL_MCBSP_IO_FSX (2)
I/O Pin Input/Output configuration for FSX Pin

**#define CSL_MCBSP_MULTICHAN_DEFAULTS **
```
{                   \
 (CSL_McbspPartMode)CSL_MCBSP_PARTMODE_2PARTITION,       \
 (CSL_McbspPartMode)CSL_MCBSP_PARTMODE_2PARTITION,       \
 (Uint16)0,                                              \
 (Uint16)0,                                              \
 (CSL_McbspPABlk)CSL_MCBSP_PABLK_0,                      \
 (CSL_McbspPBBlk)CSL_MCBSP_PBBLK_1,                      \
 (CSL_McbspPABlk)CSL_MCBSP_PABLK_0,                      \
 (CSL_McbspPBBlk)CSL_MCBSP_PBBLK_1,                      \
 }
```
Multichannel setup defaults

#define CSL_MCBSP_RFULL 0x0004
RCV full status

#define CSL_MCBSP_RRDY 0x0001
RCV ready status

#define CSL_MCBSP_RSYNCERR 0x0010
RCV frame sync error status

#define CSL_MCBSP_XEMPTY 0x0008
XMT empty status

#define CSL_MCBSP_XRDY 0x0002
XMT ready status

#define CSL_MCBSP_XSYNCERR 0x0020
XMT frame sync error status

11.6 Typedefs

typedef struct CSL_McbspObj* CSL_McbspHandle

This is a pointer to CSL_McbspObj and is passed as the first parameter to all MCBSP CSL APIs

Chapter 12
PLLC Module

Topics

12.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within PLLC module.

The primary PLL controller (PLL1) generates the input clock to the C64x+ megamodule (including the CPU) as well as most of the system peripherals such as the multichannel buffered serial ports (McBSPs) and the external memory interface (EMIF).

The PLL1 controller features a software-programmable PLL multiplier controller (PLLM) and five dividers (PREDIV, D2, D3, D4, and D5). The PLL1 controller uses the device input clock CLKIN1 to generate a system reference clock (SYSREFCLK) and four system clocks (SYSCLK2, SYSCLK3, SYSCLK4, and SYSCLK5). The divider ratio bits of dividers D2 and D3 are fixed. The divider ratio bits of dividers D4 and D5 are programmable through the PLL controller divider registers PLLDIV4 and PLLDIV5 respectively.

The secondary PLL controller generates interface clocks for the Ethernet media access controller (EMAC) and the DDR2 memory controller.

The PLL2 controller features a PLL multiplier controller and one divider (D1). The PLL multiplier is fixed and the divider D1 can be programmed through register PLLDIV1.

12.2 Functions

This section lists the functions available in the PLLC module.

12.2.1 CSL_pllcInit

CSL_Status CSL_pllcInit (CSL_PllcContext * *pContext*)

Description
This is the initialization function for the PLLC CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext  Pointer to module-context. As PLLC doesn't have any
          context based information user is expected to pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
CSL_Status          status;
...
status = CSL_pllcInit(NULL);
...
```

12.2.2 CSL_pllcOpen

CSL_PllcHandle CSL_pllcOpen (CSL_PllcObj * *pPllcObj*, CSL_InstNum *pllcNum*, CSL_PllcParam * *pPllcParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the PLLC instance and returns a handle to the instance. The handle returned by this call is input as an essential argument for rest of the APIs described for this module.

Arguments

`pPllcObj`	`Pointer to PLLC object.`
`pllcNum`	`Instance of PLLC to be opened.`
`pPllcParam`	`Pointer to module specific parameters.`
`pStatus`	`Status of the function call`

Return Value
`CSL_PllcHandle`

- Valid PLLC handle will be returned if status value is equal to CSL_SOK.

Pre Condition
CSL_pllcInit() must be called successfully before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid PLLC handle is returned
- `CSL_ESYS_FAIL` - The PLLC instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

2. PLLC object structure is populated.

Modifies
1. The status variable
2. PLLC object structure

Example

```
CSL_Status          status;
CSL_PllcObj         pllcObj;
CSL_PllcHandle      hPllc;
...
hPllc = CSL_pllcOpen(&pllcObj, CSL_PLLC_1, NULL, &status);
...
```

12.2.3 CSL_pllcClose

CSL_Status CSL_pllcClose (CSL_PllcHandle *hPllc*)

Description
This function closes the specified instance of PLLC. The device can be re-opened anytime after it has been normally closed if so required.

Arguments

`hPllc`	`Handle to the PLLC`

Return Value
`CSL_Status`

- CSL_SOK - Close successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function.

Post Condition
None

Modifies
The peripheral object structure.

Example

```
CSL_PllcHandle   hPllc;
CSL_Status       status;
...

status = CSL_pllcClose(hPllc);
...
```

12.2.4 CSL_pllcHwSetup

CSL_Status CSL_pllcHwSetup (CSL_PllcHandle *hPllc*, CSL_PllcHwSetup * *hwSetup*)

Description
It configures the PLLC registers as per the values passed in the hardware setup structure.

Arguments

```
hPllc          Handle to the PLLC

hwSetup        Pointer to hardware setup structure
```

Return Value
CSL_Status

- CSL_SOK - Hardware setup successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Hardware structure is not properly initialized

Pre Condition
Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function.

Post Condition
PLLC registers of the particular instance are configured according to the hardware setup parameters.

Modifies
PLLC registers.

Example

```
CSL_PllcHandle   hPllc;
CSL_PllcObj      pllcObj;
CSL_PllcHwSetup  hwSetup = CSL_PLLC_HWSETUP_DEFAULTS_PLL1;
CSL_Status       status;
...

hPllc = CSL_pllcOpen(&pllcObj, CSL_PLLC_1, NULL, &status);
...

hwSetup.divEnable  = (CSL_BitMask32) 0x00000001;
hwSetup.preDiv     = (Uint32)        0x00000002;
hwSetup.pllM       = (Uint32)        0x00000001;

status = CSL_pllcHwSetup(hPllc, &hwSetup);
...
```

12.2.5 CSL_pllcHwControl

CSL_Status CSL_pllcHwControl (CSL_PllcHandle ***hPllc***, CSL_PllcHwControlCmd ***cmd***, **void *** ***arg***)

Description
Takes a command of PLLC with an optional argument and implements it. This function is used to carry out the different operations performed by PLLC. For the list of commands supported and argument type that can be *void** casted and passed with a particular command refer to `CSL_PllcHwControlCmd`.

Arguments

`hPllc`	`Handle to the PLLC instance`
`cmd`	`The command to this API indicates the action to be taken on PLLC`
`arg`	`An optional argument`

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
To change PLLM, PREDIV & PLLDIVn, PLLCTL_PLLEN bit must be in BYPASS mode. Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function.

Post Condition
PLLC registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
PLLC registers determined by the command.

Example

```
CSL_PllcHandle         hPllc;
CSL_Status             status;
CSL_PllcHwControlCmd   cmd = CSL_PLLC_CMD_SET_PLLM;
Uint32   arg = 0x00000002;
...

status = CSL_pllcHwControl(hPllc, cmd, &arg);
...
```

12.2.6 CSL_pllcGetHwStatus

CSL_Status CSL_pllcGetHwStatus	**(**	**CSL_PllcHandle**	***hPllc*,**
		CSL_PllcHwStatusQuery	***query*,**
		void *	***response***
	)		

Description
Gets the status of the different operations of PLLC.

Arguments

```
hPllc          Handle to the PLLC instance

query          The query to be performed

response       Placeholder to return the status
```

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - Invalid query command
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function

Post Condition
None

Modifies
The input argument "response" is modified.

Example

```
CSL_PllcHandle          hPllc;
CSL_Status              status;
CSL_PllcHwStatusQuery   query = CSL_PLLC_QUERY_STATUS;
CSL_BitMask32           response;
...

status = CSL_pllcGetHwStatus(hPllc, query, &response);
...
```

12.2.7 CSL_pllcHwSetupRaw

CSL_Status CSL_pllcHwSetupRaw (CSL_PllcHandle *hPllc*, CSL_PllcConfig * *config*)

Description
This function initializes the device registers with the register-values provided through the config data structure. This configures registers based on a structure of register values, as compared to CSL_pllcHwSetup (), which configures registers based on structure of bit field values.

Arguments

```
hpllc        Handle to the PLLC instance

config       Pointer to config structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration is not properly initialized

Pre Condition
Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function

Post Condition
The registers of the specified PLLC instance will be setup according to input configuration structure values.

Modifies
Hardware registers of the specified PLLC instance.

Example

```
CSL_PllcHandle       hPllc;
CSL_PllcConfig       config = CSL_PLLC_CONFIG_DEFAULTS_PLL1;
CSL_Status           status;
...

status = CSL_pllcHwSetupRaw(hPllc, &config);
...
```

12.2.8 CSL_pllcGetHwSetup

CSL_Status CSL_pllcGetHwSetup	**(**	CSL_PllcHandle	***hPllc*,**
		CSL_PllcHwSetup *****	***hwSetup***
	)		

Description
It retrieves the hardware setup parameters of the PLLC specified by the given handle.

Arguments

```
hPllc        Handle to the PLLC

hwSetup      Pointer to the hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Retrieving the hardware setup parameters is successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both CSL_pllcInit() and CSL_pllcOpen() must be called successfully in order before calling this function.

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
hwSetup variable

Example

```
CSL_PllcHandle   hPllc;
CSL_PllcHwSetup  hwSetup;
CSL_Status       status;
...
status = CSL_pllcGetHwSetup(hPllc, &hwSetup);
...
```

12.2.9 CSL_pllcGetBaseAddress

CSL_Status CSL_pllcGetBaseAddress (CSL_InstNum ***pllcNum*****,**
CSL_PllcParam * ***pPllcParam*****,**
CSL_PllcBaseAddress * ***pBaseAddress***
)

Description
This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_pllcOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

pllcNum	Specifies the instance of the PLLC to be opened.
pPllcParam	Pointer to module specific parameters
pBaseAddress	Pointer to base address structure containing base address details.

Return Value
CSL_Status

- CSL_SOK - Successful on getting the base address of PLLC
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example
```
CSL_Status              status;
CSL_PllcBaseAddress     baseAddress;
...
status = CSL_pllcGetBaseAddress(CSL_PLLC_1, NULL,
                                  &baseAddress);
...
```

12.3 Data Structures

This section lists the data structures available in the PLLC module.

12.3.1 CSL_PllcObj

Detailed Description

This object contains the reference to the instance of PLLC opened using the *CSL_pllcOpen()*. The pointer to this is passed to all PLLC CSL APIs.

Field Documentation

CSL_InstNum CSL_PllcObj::pllcNum

This is the instance of PLLC being referred to by this object

CSL_PllcRegsOvly CSL_PllcObj::regs

This is a pointer to the registers of the instance of PLLC referred to by this object

12.3.2 CSL_PllcConfig

Detailed Description

Config-structure. Used to configure the PLLC using CSL_pllcHwSetupRaw(). This is a structure of register values, rather than a structure of register field values like CSL_PllcHwSetup.

Field Documentation

Uint32 CSL_PllcConfig::PLLCTL

PLL Control register. This should be configured only for PLLC instance 1

Uint32 CSL_PllcConfig::PLLDIV1

PLL Controller Divider 1 register. This should be configured only for PLLC instance 2

Uint32 CSL_PllcConfig::PLLDIV4

PLL Controller Divider 4 register. This should be configured only for PLLC instance 1

Uint32 CSL_PllcConfig::PLLDIV5

PLL Controller Divider 5 register. This should be configured only for PLLC instance 1

Uint32 CSL_PllcConfig::PLLM

PLL Multiplier Control register. This should be configured only for PLLC instance 1

Uint32 CSL_PllcConfig::PREDIV

PLL Pre-Divider Control register. This should be configured only for PLLC instance 1

12.3.3 CSL_PllcContext

Detailed Description

Module specific context information. Present implementation of PLLC CSL doesn't have any context information.

Field Documentation

Uint16 CSL_PllcContext::contextInfo

Context information of PLLC CSL. The declaration is just a placeholder for future implementation.

12.3.4 CSL_PllcHwSetup

Detailed Description
Input parameters for setting up PLL Controller. Used to put PLLC in known useful state

Field Documentation

CSL_BitMask32 CSL_PllcHwSetup::divEnable
Divider Enable/Disable.

void* CSL_PllcHwSetup::extendSetup
Setup that can be used for future implementation

Uint32 CSL_PllcHwSetup::pllDiv1
PLL Divider 1. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv4
PLL Divider 4. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllDiv5
PLL Divider 5. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllM
PLL Multiplier. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllMode
PLL Mode PLL/BYPASS. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::preDiv
Pre-Divider. This is valid only for PLLC instance 1

12.3.5 CSL_PllcParam

Detailed Description
Module specific parameters. Present implementation of PLLC CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_PllcParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

12.3.6 CSL_PllcBaseAddress

Detailed Description
This structure contains the base-address information for the peripheral instance of the PLLC.

Field Documentation

CSL_PllcRegsOvly CSL_PllcBaseAddress::regs
Base-address of the configuration registers of the peripheral

12.3.7 CSL_PllcDivRatio

Detailed Description
Input parameters for setting up PLL Divide ratio.

Field Documentation

Uint32 CSL_PllcDivRatio::divNum
Divider number

Uint32 CSL_PllcDivRatio::divRatio
Divider Ratio

12.3.8 CSL_PllcDivideControl

Detailed Description
Input parameters for enabling PLL Divide ratio

Field Documentation

CSL_PllcDivCtrl CSL_PllcDivideControl::divCtrl
Divider Control (Enable/Disable)

Uint32 CSL_PllcDivideControl::divNum
Divider Number

12.4 Enumerations

This section lists the enumerations available in the PLLC module.

12.4.1 CSL_PllcPllBypassMode

enum CSL_PllcPllBypassMode
PLLC Bypass Mode

Enumeration values:

CSL_PLLC_PLL_BYPASS_MODE	PLL Bypass Mode.
CSL_PLLC_PLL_PLL_MODE	PLL Mode.

12.4.2 CSL_PllcDivCtrl

enum CSL_PllcDivCtrl
Enums for PLL divide enable/ disable

Enumeration values:

CSL_PLLC_PLLDIV_DISABLE	PLL Divider Disable
CSL_PLLC_PLLDIV_ENABLE	PLL Divider Enable

12.4.3 CSL_PllcHwControlCmd

enum CSL_PllcHwControlCmd
This is the set of commands that are passed to the *CSL_pllcHwControl()* with an optional argument type-casted to *void** . The arguments to be passed with each enumeration (if any) are specified next to the enumeration.

Enumeration values:

CSL_PLLC_CMD_PLLCONTROL	Control PLL based on the bits set in the input argument. The least significant 16 bits of the argument should have the value to be assigned to the PLLCTL register and the most signicant 16 bits should be the value to be programmed to the PLLCMD register. **Parameters:** *(CSL_BitMask32* *)
CSL_PLLC_CMD_SET_PLLM	Set PLL multiplier value. **Parameters:** *(Uint32* *)
CSL_PLLC_CMD_SET_PLLRATIO	Set PLL divide ratio. **Parameters:** *(CSL_PllcDivRatio* *)
CSL_PLLC_CMD_PLLDIV_CONTROL	Enable/disable PLL divider. **Parameters:** *(CSL_PllcDivideControl* *)

12.4.4 CSL_PllcHwStatusQuery

enum CSL_PllcHwStatusQuery
This is used to get the status of different operations. The status is returned in the argument passed.

Enumeration values:

CSL_PLLC_QUERY_STATUS	Queries PLL Controller Status. **Parameters:** *(CSL_BitMask32*)*
CSL_PLLC_QUERY_SYSCLKSTAT	Queries PLL SYSCLK Status. **Parameters:** *(CSL_BitMask32*)*
CSL_PLLC_QUERY_RESETSTAT	Queries Reset Type Status. **Parameters:** *(CSL_BitMask32*)*

12.5 Macros

PLL Controller Status

#define CSL_PLLC_STATUS_GO   CSL_FMKT (PLLC_PLLSTAT_GOSTAT, INPROG)
Set when GO operation (divide-ratio change and clock alignment) is in progress

PLLC SYSCLK Status

#define CSL_PLLC_SYSCLKSTAT_SYS1ON   CSL_FMKT (PLLC_CKSTAT_SYS1ON, ON)
SYSCLK1 is ON

#define CSL_PLLC_SYSCLKSTAT_SYS2ON   CSL_FMKT (PLLC_CKSTAT_SYS2ON, ON)
SYSCLK2 is ON

#define CSL_PLLC_SYSCLKSTAT_SYS3ON   CSL_FMKT (PLLC_CKSTAT_SYS3ON, ON)
SYSCLK3 is ON

#define CSL_PLLC_SYSCLKSTAT_SYS4ON   CSL_FMKT (PLLC_CKSTAT_SYS4ON, ON)
SYSCLK4 is ON

#define CSL_PLLC_SYSCLKSTAT_SYS5ON   CSL_FMKT (PLLC_CKSTAT_SYS5ON, ON)
SYSCLK5 is ON

PLLC Last Reset Status

#define CSL_PLLC_RESETSTAT_MRST   CSL_FMKT (PLLC_RSTYPE_MRST, YES)
Maximum Reset

#define CSL_PLLC_RESETSTAT_POR   CSL_FMKT (PLLC_RSTYPE_POR, YES)
Power On Reset

#define CSL_PLLC_RESETSTAT_SRST   CSL_FMKT (PLLC_RSTYPE_SRST, YES)
System/Chip Reset

#define CSL_PLLC_RESETSTAT_WRST   CSL_FMKT (PLLC_RSTYPE_WRST, YES)
Warm Reset

PLLC Control Mask

#define CSL_PLLC_CTRL_BYPASS   CSL_FMKT (PLLC_PLLCTL_PLLEN, BYPASS)
PreDiv, PLL, and PostDiv are bypassed. SYSCLK divided down directly from input reference clock refclk

#define CSL_PLLC_CTRL_ENABLE   CSL_FMKT (PLLC_PLLCTL_PLLEN, PLL)
PLL is used. SYSCLK divided down from PostDiv output

#define CSL_PLLC_CTRL_MUXCTRL_PORT   CSL_FMKT (PLLC_PLLCTL_PLLENSRC, NONREGBIT)
PLLEN Mux is controlled by input pllen_pi. PLLCTL.PLLEN is don’t care

#define CSL_PLLC_CTRL_MUXCTRL_REGBIT   CSL_FMKT (PLLC_PLLCTL_PLLENSRC, REGBIT)
PLLEN Mux is controlled by PLLCTL.PLLEN. pllen_pi is don’t care

#define CSL_PLLC_CTRL_OPERATIONAL   CSL_FMKT (PLLC_PLLCTL_PLLPWRDN, NO)
Selected PLL Operational

#define CSL_PLLC_CTRL_POWERDOWN   CSL_FMKT (PLLC_PLLCTL_PLLPWRDN, YES)
Selected PLL Placed in Power Down State

#define CSL_PLLC_CTRL_RELEASE_RESET   CSL_FMKT (PLLC_PLLCTL_PLLRST, NO)
PLL Reset Released

#define CSL_PLLC_CTRL_RESET   CSL_FMKT (PLLC_PLLCTL_PLLRST, YES)
PLL Reset Asserted

PLLC Divider Enable

#define CSL_PLLC_DIVEN_PLLDIV1   (1 << 1)
Enable divider D1 for SYSCLK1

#define CSL_PLLC_DIVEN_PLLDIV4   (1 << 2)
Enable divider D4 for SYSCLK4

#define CSL_PLLC_DIVEN_PLLDIV5   (1 << 3)
Enable divider D5 for SYSCLK5

#define CSL_PLLC_DIVEN_PREDIV   (1 << 0)
PREDIV enable

Divider Select for SYSCLKs

#define CSL_PLLC_DIVSEL_PLLDIV1   (1)
Divider D1 for SYSCLK1

#define CSL_PLLC_DIVSEL_PLLDIV4 (2)
Divider D4 for SYSCLK4

#define CSL_PLLC_DIVSEL_PLLDIV5   (3)
Divider D5 for SYSCLK5

**#define CSL_PLLC_HWSETUP_DEFAULTS_PLL1 **
```
{  \
    CSL_PLLC_PLL_BYPASS_MODE,        \
    (CSL_PLLC_DIVEN_PREDIV | \
    CSL_PLLC_DIVEN_PLLDIV4 | \
    CSL_PLLC_DIVEN_PLLDIV5) ,\
    CSL_PLLC_PREDIV_RATIO_RESETVAL + 1,  \
    CSL_PLLC_PLLM_PLLM_RESETVAL  + 1,     \
    0, \
    CSL_PLLC_PLLDIV4_RATIO_RESETVAL + 1, \
    CSL_PLLC_PLLDIV5_RATIO_RESETVAL + 1, \
    NULL                             \
}
```
Default hardware setup parameters for PLL instance 1.

**#define CSL_PLLC_HWSETUP_DEFAULTS_PLL2 **
```
{  \
    0,                               \
```

```
    CSL_PLLC_DIVEN_PLLDIV1,  \
    0,                              \
    0,                              \
    CSL_PLLC_PLLDIV1_RATIO_RESETVAL + 1, \
    0,                              \
    0,                              \
    NULL                            \
}
```
Default hardware setup parameters for PLL instance 2.

**#define CSL_PLLC_CONFIG_DEFAULTS_PLL1 **
```
{    \
    CSL_PLLC_PLLCTL_RESETVAL,        \
    CSL_PLLC_PLLM_RESETVAL,          \
    CSL_PLLC_PREDIV_RESETVAL,        \
    0,        \
    CSL_PLLC_PLLDIV4_RESETVAL,       \
    CSL_PLLC_PLLDIV5_RESETVAL        \
}
```
Default values for config structure for PLL instance 1.

**#define CSL_PLLC_CONFIG_DEFAULTS_PLL2 **
```
{    \
    0,                                      \
    0,                                      \
    0,                                      \
    CSL_PLLC_PLLDIV1_RESETVAL,  \
    0,                                      \
    0                                       \
}
```
Default values for config structure for PLL instance 2.

**#define CSL_PLLC_ HWSETUP_DEFAULTS_750MHZ **
```
{  \
    CSL_PLLC_PLL_PLL_MODE,          \
    0,                              \
    0,                              \
    15,                             \
    0,                              \
    0,                              \
    0,                              \
    NULL                            \
}
```
Default hardware setup parameters for output clock frequency of 750 MHz , CLKIN = 50MHz

**#define CSL_PLLC_ HWSETUP_DEFAULTS_1GHZ **

```
{  \
    CSL_PLLC_PLL_PLL_MODE,           \
    0,                               \
    0,                               \
    20,                              \
    0,                               \
    0,                               \
    0,                               \
    NULL                             \
}
```

Default hardware setup parameters for output clock frequency of 1 GHz , CLKIN = 50MHz

12.6 Typedefs

typedef CSL_PllcObj * CSL_PllcHandle
This data type is used to return the handle to the pllc functions.

Chapter 13
SRIO Module

Topics

13.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within SRIO module.

RapidIO™ is a non-proprietary high-bandwidth system level interconnect, it is a packet-switched interconnect intended primarily as an intra-system interface for chip-to-chip and board-to-board communications at Gigabyte-per-second performance levels. Uses for the architecture can be found in connected microprocessors, memory, and memory mapped I/O devices that operate in networking equipment, memory subsystems, and general purpose computing.

Features Supported in SRIO:

- RapidIO Interconnect Specification V1.2 compliant, Errata 1.2
- LP-Serial Specification V1.2 compliant
- 4X Serial RapidIO with auto-negotiation to 1X port, optional operation for (4) 1X ports
- Integrated Clock Recovery with TI SERDES
- Hardware Error handling including CRC
- Differential CML signaling supporting AC and DC coupling
- Support for 1.25, 2.5, and 3.125Gbps rates
- Power-down option for unused ports
- Read, write, write w/response, streaming write, out-going Atomic, maintenance operations
- Shall generate interrupts to the CPU (Doorbell packets and Internal scheduling)
- Support for 8b and 16b device ID
- Support for receiving 34b addresses
- Support for generating 34b, 50b, and 66b addresses
- Support for data sizes: byte, half-word, word, double-word
- Defined as Big Endian
- Direct IO transfers
- Message passing transfers
- Data payloads to 256B
- Single message generation up to 16 packets
- Elastic Store FIFO for clock domain handoff
- Short Run and Long Run compliant
- CBA3.0 compliant – generate DMA BUS commands and data transfers
- Support for Error Management Extensions
- Support for Congestion Control Extensions
- Support for one multi-cast ID

The SRIO CSL supports functional layer API doesnot support the functional layer API for message passing data transfer.

13.2 Functions

This section lists the functions available in the SRIO module.

13.2.1 CSL_srioInit

CSL_Status CSL_srioInit (CSL_SrioContext * *pContext*)

Description

This is the initialization function for the SRIO CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context. As SRIO doesn't
            have any context based information user is expected
            to pass NULL.
```

Return Value

`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition

None

Post Condition

The CSL for SRIO is initialized

Modifies

None

Example

```
CSL_Status          status;
...
status = CSL_srioInit(NULL);
...
```

13.2.2 CSL_srioOpen

CSL_SrioHandle CSL_srioOpen (CSL_SrioObj * *pSrioObj*, CSL_InstNum *srioNum*, CSL_SrioParam * *pSrioParam*, CSL_Status * *pStatus*)

Description

This function populates the peripheral data object for the SRIO instance and returns a handle to the instance. The handle returned by this call is input as an essential argument for the rest of the APIs described for this module.

Arguments

```
pSrioObj          Pointer to SRIO object.

srioNum           Instance of SRIO CSL to be opened.
                  There is one instance of the SRIO
                  available. So, the value for this parameter will
                  be CSL_SRIO always.

pSrioParam        Module specific parameters.

pStatus           Status of the function call
```

Return Value
`CSL_SrioHandle`

- Valid SRIO handle will be returned if status value is equal to CSL_SOK. Otherwise NULL is returned

Pre Condition
The SRIO must be successfully initialized via CSL_*srioInit* () before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid SRIO handle is returned
- `CSL_ESYS_FAIL` - The SRIO instance is invalid
- `CSL_ESYS_INVPARAMS` – Invalid parameter

2. SRIO object structure is populated.

Modifies
1. The status variable
2. SRIO object structure

Example

```
CSL_Status     status;
CSL_SrioObj    srioObj;
CSL_SrioHandle hSrio;
...
hSrio = CSL_srioOpen(&srioObj, CSL_SRIO, NULL, &status);
...
```

13.2.3 CSL_srioClose

CSL_Status CSL_srioClose (CSL_SrioHandle *hSrio*)

Description
This function closes the specified instance of SRIO.

Arguments

```
hSrio           Handle to the SRIO
```

Return Value
`CSL_Status`

- `CSL_SOK` - SRIO is closed successfully
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling *CSL_srioClose()*.

Post Condition
The SRIO CSL APIs can not be called until the SRIO CSL is reopened again using *CSL_srioOpen()*.

Modifies
The peripheral data object.

Example

```
CSL_SrioHandle hSrio;
CSL_Status     status;
...
status = CSL_srioClose(hSrio);
...
```

13.2.4 CSL_srioHwSetup

CSL_Status CSL_srioHwSetup **(** CSL_SrioHandle ***hSrio***, CSL_SrioHwSetup ****hwSetup*** **)**

Description
It configures the SRIO instance registers as per the values passed in the hardware setup structure.

Arguments

```
hSrio           Handle to the SRIO instance

hwSetup         Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Hardware structure is not properly initialized

Pre Condition

Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API.

Post Condition
The specified instance will be setup according to value passed.

Modifies
Hardware registers for the specified instance.

Example

```
CSL_SrioHandle          hSrio;
CSL_SrioObj             srioObj;
CSL_SrioHwSetup         hwSetup;
CSL_Status              status;
Uint8                   index = 0;

hwSetup.perEn = TRUE;
// Enable loopback operation
hwSetup->periCntlSetup.loopback = 1;

hwSetup->periCntlSetup.bootComplete = 1;

// Enable clocks to all domains
hwSetup->gblEn = 1;

for (index=0; index<9; index++) { /* 9 domains */
  hwSetup->blkEn[index] = 1;    /* Enable each of it */
}
...
hSrio = CSL_srioOpen(&srioObj, CSL_SRIO, NULL, &status);
status = CSL_srioHwSetup(hSrio, &hwSetup);
...
```

13.2.5 CSL_srioHwControl

CSL_Status CSL_srioHwControl	**(**	CSL_SrioHandle	***hSrio*,**
		CSL_SrioHwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
This function performs various control operations on the SRIO instance, based on the command passed.

Arguments

```
hSrio       Handle to the SRIO instance

cmd         Operation to be performed on the SRIO

arg         Argument specific to the command
```

Return Value
`CSL_Status`

- `CSL_SOK` - Command execution successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command
- CSL_ESYS_INVPARAMS – Invalid parameters

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API.

Post Condition
Registers of the SRIO instance are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
Registers determined by the command

Example

```
CSL_SrioHandle   hSrio;
CSL_SrioPortData clearData;
Uint32           mask;
Uint8            index;
...
// for clearing LSU interrupts status [0..3]
index = 1;
mask = CSL_SRIO_LSU_INTR3 | CSL_SRIO_LSU_INTR2 |
       CSL_SRIO_LSU_INTR1 | CSL_SRIO_LSU_INTR0;
clearData.index = index;
clearData.data = mask;
...
CSL_srioHwControl(hSrio, CSL_SRIO_CMD_LSU_INTR_CLEAR, &clearData);
...
```

13.2.6 CSL_srioGetHwStatus

CSL_Status CSL_srioGetHwStatus	**(**	[CSL_SrioHandle](#)	***hSrio*,**
		[CSL_SrioHwStatusQuery](#)	***query*,**
		void *	***response***
	)		

Description
This function is used to get the value of various parameters of the SRIO instance. The value returned depends on the query passed.

Arguments

```
hSrio           Handle to the SRIO instance
```

```
query          Query to be performed

response       Pointer to buffer to return the data requested by
               the query passed
```

Return Value
CSL_Status

- CSL_SOK - Successful completion of the query
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Query command not supported
- CSL_ESYS_INVPARAMS – Invalid parameter

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API

Post Condition
None

Modifies
The input argument "response" is modified.

Example

```
CSL_Status        status;
CSL_SrioHandle    hSrio;
CSL_SrioPidNumber response;
...
status=CSL_srioGetHwStatus(hSrio, CSL_SRIO_QUERY_PID_NUMBER,
                              &response);
...
```

13.2.7 CSL_srioHwSetupRaw

CSL_Status CSL_srioHwSetupRaw (CSL_SrioHandle *hSrio*, CSL_SrioConfig * *config*)

Description
This function initializes the device registers with the register-values provided through the config data structure. This configures registers based on a structure of register values, as compared to CSL_SrioHwSetup, which configures registers based on structure of bit field values.

Arguments

```
hSrio          Handle to the SRIO instance

config         Pointer to the config structure containing the
               device register values
```

Return Value

CSL_Status

- CSL_SOK - Configuration successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Configuration structure pointer is not properly initialized

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API

Post Condition
The registers of SRIO will be setup according to the values passed through the config structure.

Modifies
Hardware registers of SRIO

Example

```
CSL_SrioHandle hSrio;
CSL_SrioConfig config = CSL_SRIO_CONFIG_DEFAULTS;
CSL_Status     status;
...

status = CSL_srioHwSetupRaw(hSrio, &config);
...
```

13.2.8 CSL_srioGetHwSetup

CSL_Status CSL_srioGetHwSetup	**(**	**CSL_SrioHandle**	***hSrio*,**
		CSL_SrioHwSetup *	***hwSetup***
	)		

Description
It retrieves the hardware setup parameters.

Arguments

hSrio	Handle to the SRIO instance
hwSetup	Pointer to hardware setup structure

Return Value
CSL_Status

- CSL_SOK - Hardware setup retrieved
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS – Invalid parameter

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API

Post Condition
The hardware set up structure will be populated with values from the registers.

Modifies
None

Example

```
CSL_SrioHandle      hSrio;
CSL_Status          status;
CSL_SrioHwSetup     hwSetup;
...
status = CSL_srioGetHwSetup(hSrio, &hwSetup);
...
```

13.2.9 CSL_srioLsuSetup

CSL_Status CSL_srioLsuSetup	**(**	**CSL_SrioHandle**	***hSrio*,**
		CSL_SrioDirectIO_ConfigXfr *	***lsuConfig*,**
		Uint8	***index***
	)		

Description
Function to configure the LSU module for Direct IO transfer.

Arguments

hSrio	Handle to the SRIO instance
lsuConfig	Pointer to the direct IO configuration structure
index	Index to the LSU block number

Return Value
CSL_Status

- CSL_SOK - Successfully configured the LSU module
- CSL_ESYS_BADHANDLE - Invalid handle is passed
- CSL_ESYS_INVPARAMS – Invalid parameter

Pre Condition
Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API

Post Condition
The LSU module registers are configured with the passed parameters and the data transfer starts.

Modifies
LSU module registers

Example

```
extern Uint8                        *src;
```

```
extern Uint8                            *dst;

CSL_SrioHandle                          hSrio;
CSL_Status                              status;
CSL_SrioDirectIO_ConfigXfr              lsuConfig;
Uint8                                   index;

index = 1;
...

lsuConfig.srcNodeAddr         = (Uint32)src; /* Source address */
lsuConfig.dstNodeAddr.addressHi = 0;
lsuConfig.dstNodeAddr.addressLo = (Uint32)dst; /* Destination
                                                  address */
lsuConfig.byteCnt          = 256;
lsuConfig.idSize           = 1;       /* 16 bit device id */
lsuConfig.priority         = 2;      /* PKT priority is 2 */
lsuConfig.xambs            = 0; /*Not an extended address */
lsuConfig.dstId            = 0xABCD;
lsuConfig.intrReq          = 0;         /* No interrupts */
lsuConfig.pktType          = 0x54; /* write with no response */
lsuConfig.hopCount         = 0; /*Valid for maintainance pkt */
lsuConfig.doorbellInfo     = 0;    /* Not a doorbell pkt */
lsuConfig.outPortId       = 3;  /* Tx on Port SELECTED_PORT */
status = CSL_srioLsuSetup(hSrio, &lsuConfig, index);
...
```

13.2.10 CSL_srioGetBaseAddress

CSL_Status CSL_srioGetBaseAddress **(CSL_InstNum** ***srioNum*****,**
CSL_SrioParam * ***pSrioParam*****,**
CSL_SrioBaseAddress * ***pBaseAddress***
)

Description
This function gets the base address of the given SRIO instance. This function will be called inside the CSL_srioOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral. MMRs go to an alternate location.

Arguments

```
srioNum        Specifies the instance of the SRIO to be opened

pSrioParam     SRIO module specific parameters

pBaseAddress   Pointer to base address structure containing base
               address details
```

Return Value
`CSL_Status`

- `CSL_SOK` - Successfull on getting the base address of SRIO.

- CSL_ESYS_FAIL - SRIO instance is not available.
- CSL_ESYS_INVPARAMS - Invalid Parameters

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
Base address structure is modified.

Example

```
CSL_Status          status;
CSL_SrioBaseAddress baseAddress;
...
status = CSL_SrioGetBaseAddress(CSL_SRIO, NULL, &baseAddress);
...
```

13.3 Data Structures

This section lists the data structures available in the SRIO module.

13.3.1 CSL_SrioObj

Detailed Description
Serial Rapid IO object structure.

Field Documentation

CSL_InstNum CSL_SrioObj::perNum
Instance of SRIO being referred by this object

CSL_SrioRegsOvly CSL_SrioObj::regs
Pointer to the register overlay structure of the SRIO

13.3.2 CSL_SrioConfig

Detailed Description
Config-structure used to configure the SRIO using CSL_srioHwSetupRaw(). This is a structure of register values, rather than a structure of register field values like CSLSrioHwSetup

Field Documentation

Uint32 CSL_SrioConfig::BASE_ID
Base device ID CSR register

Uint32 CSL_SrioConfig::BLK_EN[9]
Block enable registers

Uint32 CSL_SrioConfig::COMP_TAG
Component tag CSR

Uint32 CSL_SrioConfig::DEVICEID_REG1
Device ID register 1

Uint32 CSL_SrioConfig::DEVICEID_REG2
Device ID register 2

Uint32 CSL_SrioConfig::DOORBELL_ICCR[4]
Doorbell interrupt clear registers

Uint32 CSL_SrioConfig::ERR_DET
Logical/Transport layer error detect CSR

Uint32 CSL_SrioConfig::ERR_EN
Logical/Transport layer error enable CSR

Uint32 CSL_SrioConfig::ERR_RST_EVNT_ICCR
Error, Reset, and Special event interrupt clear registers

Uint32 CSL_SrioConfig::FLOW_CNTL[16]
Flow control table entry registers

Uint32 CSL_SrioConfig::GBL_EN
Peripheral global enable register

Uint32 CSL_SrioConfig::HOST_BASE_ID_LOCK
Host base device ID lock CSR

CSL_SrioHw_pkt_fwdRegs CSL_SrioConfig::HW_PKT_FWD[4]
Packet forwarding registers for 16-bit and 8-bit device IDs

Uint32 CSL_SrioConfig::INTDST_RATE_CNTL
INTDST interrupt rate control register for DST 0

Uint32 CSL_SrioConfig::IP_PRESCALAR
Serial port IP prescalar

CSL_SrioCfgLsuRegs CSL_SrioConfig::LSU[4]
LSU registers

Uint32 CSL_SrioConfig::LSU_ICCR
LSU interrupt clear registers

Uint32 CSL_SrioConfig::PCR
Peripheral control register

Uint32 CSL_SrioConfig::PE_LL_CTL
Processing element logical layer control CSR register

Uint32 CSL_SrioConfig::PER_SET_CNTL
Peripheral settings control register

CSL_SrioCfgPortRegs CSL_SrioConfig::PORT[4]
Port registers

CSL_SrioCfgPortErrorRegs CSL_SrioConfig::PORT_ERROR[4]
Port error CSR

CSL_SrioCfgPortOptionRegs CSL_SrioConfig::PORT_OPTION[4]
Port options CSR

Uint32 CSL_SrioConfig::PW_TGT_ID
Port-write target device ID CSR

Uint32 CSL_SrioConfig::SERDES_CFG_CNTL[4]
SerDes macros configuration control registers

Uint32 CSL_SrioConfig::SERDES_CFGRX_CNTL[4]
SerDes RX channels configuration control registers

Uint32 CSL_SrioConfig::SERDES_CFGTX_CNTL[4]
SerDes TX channels configuration control registers

Uint32 CSL_SrioConfig::SP_GEN_CTL
Port general control CSR

Uint32 CSL_SrioConfig::SP_IP_DISCOVERY_TIMER
Port IP discovery timer in 4x mode

Uint32 CSL_SrioConfig::SP_IP_MODE
Port IP mode CSR

Uint32 CSL_SrioConfig::SP_LT_CTL
Port link time-out control CSR

Uint32 CSL_SrioConfig::SP_RT_CTL
Port link response time-out control CSR

13.3.3 CSL_SrioContext

Detailed Description
Module specific context information. Present implementation of SRIO CSL doesn't have any context information.

Field Documentation

Uint16 CSL_SrioContext::contextInfo
Context information of SRIO CSL. The declaration is just a placeholder for future implementation.

13.3.4 CSL_SrioHwSetup

Detailed Description
Hardware setup structure.

Field Documentation

Uint32 CSL_SrioHwSetup::blkEn[9]
Controls reset to logical block n

Uint32 CSL_SrioHwSetup::componentTag
Software defined component Tag for PE (processing element). Useful for devices without device IDs

Uint32 CSL_SrioHwSetup::deviceId1
This value is equal to the value of the RapidIO Base Device ID CSR. The CPU must read the CSR value and set this register, so that out-going packets contain the correct SOURCEID value. This field contains both 16bit and 8bit IDs

Uint32 CSL_SrioHwSetup::deviceId2
This is a secondary supported DeviceID checked against an in-coming packet's DestID field. Typically used for Multi-cast support. This field contains both 16bit and 8bit IDs

CSL_SrioDevIdConfig CSL_SrioHwSetup::devIdSetup
Base device configuration

CSL_SrioDiscoveryTimer CSL_SrioHwSetup::discoveryTimer
Discovery Timer in 4x mode. The discovery-timer allows time for the link partner to enter its DISCOVERY state and if the link partner is supporting 4x mode, for all 4 lanes to be aligned

Uint16 CSL_SrioHwSetup::flowCntlId[16]
Destination ID of flow n

Uint8 CSL_SrioHwSetup::flowCntlIdLen[16]
Selects flow control ID length

Bool CSL_SrioHwSetup::gblEn
Controls reset to all clock domains within the peripheral

Uint32 CSL_SrioHwSetup::lgclTransErrEn
Enable/disable logical/transport layer errors. Macros can be OR'ed to get the value to pass the argument

CSL_SrioAddrSelect CSL_SrioHwSetup::peLlAddrCtrl
Sets the number of address bits generated by the PE as a source and processed by the PE as the target of an operation

Bool CSL_SrioHwSetup::perEn
Peripheral enable. Controls the flow of data in the logical layer of the peripheral

CSL_SrioControlSetup CSL_SrioHwSetup::periCntlSetup
This is used to hold the information for local SRIO's control setup

CSL_SrioPktFwdCntl CSL_SrioHwSetup::pktFwdCntl[4]
Sets the boundaries for device IDs that are part of the chain and the packet can be forwarded to

Uint32 CSL_SrioHwSetup::portCntlIndpEn[4]
Port control independent error reporting enable. Macros can be OR'ed to get the value

CSL_SrioPortCntlConfig CSL_SrioHwSetup::portCntlSetup[4]
Port control configuration

CSL_SrioPortErrConfig CSL_SrioHwSetup::portErrSetup[4]
Port error configuration

CSL_SrioPortGenConfig CSL_SrioHwSetup::portGenSetup
Port General configuration

Uint32 CSL_SrioHwSetup::portIpModeSet
This configures the SP_IP_MODE register

Uint32 CSL_SrioHwSetup::portIpPrescalar
This configures the SP_IP_PRESCALE register

CSL_SrioPwTimer CSL_SrioHwSetup::pwTimer
Port-Write Timer. The timer defines a period to repeat sending an error reporting Port-Write request for software assistance. The timer is stopped by software writing to the error detect registers.

CSL_SrioSerDesPllCfg CSL_SrioHwSetup::serDesPllCfg[4]
General Purpose I/O bits can be used to control any SerDes PLL control functions. Mapping of GPIO bits is device specific based on the SERDES macro that is implemented

CSL_SrioSerDesRxCfg CSL_SrioHwSetup::serDesRxChannelCfg[4]
SERDES RX channel configure

CSL_SrioSerDesTxCfg CSL_SrioHwSetup::serDesTxChannelCfg[4]

SERDES TX channel configure

CSL_SrioSilenceTimer CSL_SrioHwSetup::silenceTimer[4]
Silence timer. Defines the time of the port in the SILENT state

13.3.5 CSL_SrioParam

Detailed Description
Module specific parameters. Present implementation of SRIO CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_SrioParam::flags
Bit mask to be used for module specific parameters. The declaration is just a place-holder for future implementation.

13.3.6 CSL_SrioBaseAddress

Detailed Description
This structure contains the base-address information for the peripheral instance.

Field Documentation

CSL_SrioRegsOvly CSL_SrioBaseAddress::regs
Base-address of the configuration registers of the peripheral

13.3.7 CSL_SrioCfgLsuRegs

Detailed Description
This structure contains the control and congestion flow mask registers for the configuration of Load/Store module in SRIO.

Field Documentation

Uint32 CSL_SrioCfgLsuRegs::LSU_FLOW_MASKS
Core LSU congestion control flow mask register

Uint32 CSL_SrioCfgLsuRegs::LSU_REG0
LSU control register 0

Uint32 CSL_SrioCfgLsuRegs::LSU_REG1
LSU control register 1

Uint32 CSL_SrioCfgLsuRegs::LSU_REG2
LSU control register 2

Uint32 CSL_SrioCfgLsuRegs::LSU_REG3
LSU control register 3

Uint32 CSL_SrioCfgLsuRegs::LSU_REG4
LSU control register 4

13.3.8 CSL_SrioCfgPortRegs

Detailed Description

This structure contains port configuration CSR registers.
Field Documentation

Uint32 CSL_SrioCfgPortRegs::SP_ACKID_STAT
Port local ACK ID status CSR

Uint32 CSL_SrioCfgPortRegs::SP_CTL
Port control CSR

Uint32 CSL_SrioCfgPortRegs::SP_ERR_STAT
Port error and status CSR

Uint32 CSL_SrioCfgPortRegs::SP_LM_REQ
Port link maintenance request CSR

13.3.9 CSL_SrioCfgPortErrorRegs

Detailed Description
This structure contains port error configuration CSR registers.

Field Documentation

Uint32 CSL_SrioCfgPortErrorRegs::SP_ERR_DET
Port error detect CSR

Uint32 CSL_SrioCfgPortErrorRegs::SP_ERR_RATE
Port error rate CSR

Uint32 CSL_SrioCfgPortErrorRegs::SP_ERR_THRESH
Port error rate threshold CSR

Uint32 CSL_SrioCfgPortErrorRegs::SP_RATE_EN
Port error enable CSR

13.3.10 CSL_SrioCfgPortOptionRegs

Detailed Description
This structure contains port error configuration CSR registers.

Field Documentation

Uint32 CSL_SrioCfgPortOptionRegs::SP_CS_TX
Port control symbol transmit register

Uint32 CSL_SrioCfgPortOptionRegs::SP_CTL_INDEP
Port control independent register

Uint32 CSL_SrioCfgPortOptionRegs::SP_MULT_EVNT_CS
Port multicast-event control symbol request register

Uint32 CSL_SrioCfgPortOptionRegs::SP_RST_OPT
Port reset option CSR

Uint32 CSL_SrioCfgPortOptionRegs::SP_SILENCE_TIMER
Port silence timer register

13.3.11 CSL_SrioControlSetup

Detailed Description
This structure contains the control parameters of SRIO.

Field Documentation

Bool CSL_SrioControlSetup::bootComplete
Controls ability to write any register during initialization. It also includes read only registers during normal mode of operation that have application defined reset value. 0 - write enabled, 1 - write to read only registers disabled. Usually the boot_complete is asserted once after reset to define power on configuration

CSL_SrioBufMode CSL_SrioControlSetup::bufferMode
UDI buffering setup (priority versus port)

CSL_SrioBusTransPriority CSL_SrioControlSetup::busTransPriority
Internal bus transaction priority

Bool CSL_SrioControlSetup::loopback
0 - Normal operation, 1 - Loop back. Transmit data to receive on the same port. Packet data is looped back in the digital domain before the SerDes macros

Uint8 CSL_SrioControlSetup::pllEn
SERDES macros PLL enable/disable. Enable/disable macros are OR' ed to get the value

CSL_SrioClkDiv CSL_SrioControlSetup::prescalar
Internal clock frequency pre-scalar, used to drive the request to response timers

Bool CSL_SrioControlSetup::swMemSleepOverride
Puts the memories in either in sleep mode or in awake mode, while in shutdown

CSL_SrioTxPriorityWm CSL_SrioControlSetup::txPriority0Wm
Sets the required number of logical layer TX buffers needed to send priority 0 packets across the UDI interface

CSL_SrioTxPriorityWm CSL_SrioControlSetup::txPriority1Wm
Sets the required number of logical layer TX buffers needed to send priority 1 packets across the UDI interface

CSL_SrioTxPriorityWm CSL_SrioControlSetup::txPriority2Wm
Sets the required number of logical layer TX buffers needed to send priority 2 packets across the UDI interface

13.3.12 CSL_SrioDevInfo

Detailed Description
This structure contains SRIO vendor related information.

Field Documentation

Uint16 CSL_SrioDevInfo::devId
Identifies the vendor specific type of device

Uint32 CSL_SrioDevInfo::devRevision
Vendor supplied device revision

Uint16 CSL_SrioDevInfo::devVendorId
Device vendor ID assigned by RapidIO[TA]

13.3.13 CSL_SrioAssyInfo

Detailed Description
This structure contains the information about SRIO assembly.

Field Documentation

Uint16 CSL_SrioAssyInfo::assyId
Identifies the vendor specific type of assembly

Uint16 CSL_SrioAssyInfo::assyRevision
Vendor supplied assembly revision

Uint16 CSL_SrioAssyInfo::assyVendorId
Assembly vendor ID assigned by RapidIO TA

13.3.14 CSL_SrioCntlSym

Detailed Description
This structure contains control symbols used for packet acknowledgment.

Field Documentation

Uint8 CSL_SrioCntlSym::cmd
Used in conjunction with stype1 encoding to define the link maintenance commands

Bool CSL_SrioCntlSym::emb
When set, force the outbound flow to insert control symbol into packet. Used in debug mode

Uint8 CSL_SrioCntlSym::par0
Used in conjunction with stype0 encoding

Uint8 CSL_SrioCntlSym::par1
Used in conjunction with stype0 encoding

Uint8 CSL_SrioCntlSym::stype0
Encoding for control symbol that make use of parameters PAR_0 and PAR_1

Uint8 CSL_SrioCntlSym::stype1
Encoding for control symbol that make use of parameter CMD

CSL_SrioPortNum CSL_SrioCntlSym:: portNum
Port number

13.3.15 CSL_SrioLogTrErrInfo

Detailed Description
This structure contains captured error information of logical/transport layer.

Field Documentation

Uint16 CSL_SrioLogTrErrInfo::destId
The destination ID associated with the error

Uint32 CSL_SrioLogTrErrInfo::errAddrHi
The address associated with the error (only valid for devices supporting 66 and 50 bit addresses)

Uint32 CSL_SrioLogTrErrInfo::errAddrLo
The address associated with the error (only valid for devices supporting 66 and 50 bit addresses)

Uint8 CSL_SrioLogTrErrInfo::ftype
Format type associated with the error

Uint16 CSL_SrioLogTrErrInfo::impSpecific
Implementation specific information associated with the error

Uint16 CSL_SrioLogTrErrInfo::srcId
The source ID associated with the error

Uint8 CSL_SrioLogTrErrInfo::tType
Transaction type associated with the error

Uint8 CSL_SrioLogTrErrInfo::xambs
Extended address bits of the address associated with the error

13.3.16 CSL_SrioPortData

Detailed Description
This structure is used to hold the configuration/status information of different SRIO ports.

Field Documentation

CSL_BitMask32 CSL_SrioPortData::data
Desired information in the registers

Uint32 CSL_SrioPortData::index
Port selection

13.3.17 CSL_SrioPortGenConfig

Detailed Description
This structure contains information to configure port.

Field Documentation

Bool CSL_SrioPortGenConfig::hostEn
A Host device enable 0b0 - agent or slave device 0b1 - host device

Bool CSL_SrioPortGenConfig::masterEn
It controls whether or not a device is allowed to issue requests into the system. If the Master Enable is not set, the device may only respond to requests

Uint32 CSL_SrioPortGenConfig::portLinkTimeout
Timeout value for all ports on the device. This timeout is for link events such as sending a packet to receiving the corresponding ACK

Uint32 CSL_SrioPortGenConfig::portRespTimeout
Timeout value for all ports on the device. This timeout is for sending a packet to receiving the corresponding response packet

13.3.18 CSL_SrioPortCntlConfig

Detailed Description
This structure contains information to configure port parameters.

Field Documentation

Bool CSL_SrioPortCntlConfig::dropPktEn
Enabling this bit causes the port to drop packets that are acknowledged with a packet-not-accepted control symbol when the error failed threshold is exceeded

Bool CSL_SrioPortCntlConfig::errCheckDis
Disables/Enables all RapidIO transmission error checking

Bool CSL_SrioPortCntlConfig::inPortEn
Input port receive enable. Controls input port to respond to any packet

Bool CSL_SrioPortCntlConfig::multicastRcvEn
Disables/Enables the multicast event reception on this port

Bool CSL_SrioPortCntlConfig::outPortEn
Controls output port to issue any packets and control symbols

Bool CSL_SrioPortCntlConfig::portDis
Controls port receivers/drivers to receive/transmit to any packets or control symbols

Bool CSL_SrioPortCntlConfig::portLockoutEn
When the bit is set the port is stopped and is not enabled to issue or receive any packets

CSL_SrioPortWidthOverride CSL_SrioPortCntlConfig::portWidthOverride
Soft port configuration to override the hardware size

Bool CSL_SrioPortCntlConfig::stopOnPortFailEn
Enabling this bit causes the port to stop attempting to send packets to the connected device when the output failed-encountered bit is set.

13.3.19 CSL_SrioPortErrConfig

Detailed Description
This structure contains information to configure port error enable and error rate thresholds.

Field Documentation

Uint32 CSL_SrioPortErrConfig::portErrRateEn
Enable/disable port error interrupts. Macros can be OR'ed to get the value to pass the argument

Uint8 CSL_SrioPortErrConfig::portErrRtDegrdThresh
The threshold value for reporting an error condition due to a degrading link

Uint8 CSL_SrioPortErrConfig::portErrRtFldThresh
The threshold value for reporting an error condition due to a possibly broken link

CSL_SrioErrRtNum **CSL_SrioPortErrConfig::portErrRtRec**
Limit value to the error rate counter above the failed threshold trigger

CSL_SrioErrRtBias **CSL_SrioPortErrConfig::prtErrRtBias**
The error rate bias value

13.3.20 CSL_SrioPidNumber

Detailed Description
This structure is used to return the contents of the Peripheral Identification register, which has the versioning information, used to identify the specific SRIO peripheral.

Field Documentation

Uint8 CSL_SrioPidNumber::srioClass
Identifies the class of peripheral

Uint8 CSL_SrioPidNumber::srioRevision
Identifies the revision of SRIO

Uint8 CSL_SrioPidNumber::srioType
Identifies the type of peripheral

13.3.21 CSL_SrioDevIdConfig

Detailed Description
This structure contains base device configuration parameters.

Field Documentation

Uint16 CSL_SrioDevIdConfig::hostBaseDevId
This is the base ID for the Host PE that is initializing this PE (processing element)

Uint16 CSL_SrioDevIdConfig::largeTrBaseDevId
This is the base ID of the device in a large common transport system (Only valid for end points, and if bit 4 of the PEFTR register is set)

Uint8 CSL_SrioDevIdConfig::smallTrBaseDevId
This is the base ID of the device in small common transport system (End points only)

13.3.22 CSL_SrioBlkEn

Detailed Description
This structure is used to enable/disable the blocks within the SRIO peripheral.

Field Documentation

Bool CSL_SrioBlkEn::block0
Enable/disable MMR non-Reset/PD control Registers (Logical Block 0)

Bool CSL_SrioBlkEn::block1
Enable/disable LSU (Direct I/O Initiator)

Bool CSL_SrioBlkEn::block2
Enable/disable MAU (Direct I/O Target)

Bool CSL_SrioBlkEn::block3
Enable/disable TXU (Message Passing Initiator)

Bool CSL_SrioBlkEn::block4
Enable/disable RXU (Message Passing Target)

Bool CSL_SrioBlkEn::block5
Enable/disable Port 0 Data path

Bool CSL_SrioBlkEn::block6
Enable/disable Port 1 Data path

Bool CSL_SrioBlkEn::block7
Enable/disable port 2 Data path

Bool CSL_SrioBlkEn::block8
Enable/disable Port 3 Data path

13.3.23 CSL_SrioPktFwdCntl

Detailed Description
This structure is used to configure hardware packet forwarding.

Field Documentation

Uint16 CSL_SrioPktFwdCntl::largeLowBoundDevId
Lower 16-bit Device ID boundary. Destination ID lower than this number cannot use the table entry

Uint16 CSL_SrioPktFwdCntl::largeUpBoundDevId
Upper 16-bit Device ID boundary. Destination ID above this range cannot use the table entry

CSL_SrioPortNum CSL_SrioPktFwdCntl::outBoundPort
Output port number for packet's whose destination ID falls within the 8b or 16b range for this table entry

Uint8 CSL_SrioPktFwdCntl::smallLowBoundDevId
Lower 8-bit Device ID boundary. Destination ID lower than this number cannot use the table entry

Uint8 CSL_SrioPktFwdCntl::smallUpBoundDevId
Upper 8-bit Device ID boundary. Destination ID above this range cannot use the table entry

13.3.24 CSL_SrioLsuCompStat

Detailed Description
This structure is used to return the completion status of the LSU command.

Field Documentation

CSL_SrioCompCode CSL_SrioLsuCompStat::lsuCompCode
This is used to return the LSU command completion code

CSL_SrioPortNum CSL_SrioLsuCompStat::portNum
Port number

13.3.25 CSL_SrioLongAddress

Detailed Description
This structure contains local configuration base address.

Field Documentation

Uint32 CSL_SrioLongAddress::addressHi
Configuration address high

Uint32 CSL_SrioLongAddress::addressLo
Configuration address low

13.3.26 CSL_SrioPortErrCapt

Detailed Description
This structure is used to return the error capture information for the specified port.

Field Documentation

Uint32 CSL_SrioPortErrCapt::capture0
This contains the control symbol information or 0-3 bytes of packet header

Uint32 CSL_SrioPortErrCapt::capture1
This contains the control symbol information or 4-7 bytes of packet header

Uint32 CSL_SrioPortErrCapt::capture2
This contains the control symbol information or 8-11 bytes of packet header

Uint32 CSL_SrioPortErrCapt::capture3
This contains the control symbol information or 12-15 bytes of packet header

Uint8 CSL_SrioPortErrCapt::errorType
Encoded error type

Uint32 CSL_SrioPortErrCapt::impSpecData
Implementation specific data

CSL_SrioPortCaptType CSL_SrioPortErrCapt::portErrCaptType
Type of information logged

CSL_SrioPortNum CSL_SrioPortErrCapt::portNum
Port number for which the error data is to be captured

13.3.27 CSL_SrioPortWriteCapt

Detailed Description
This structure is used to return the port write capture information.

Field Documentation

Uint32 CSL_SrioPortWriteCapt::capture0
Port-Write payload, word 0

Uint32 CSL_SrioPortWriteCapt::capture1
Port-Write payload, word 1

Uint32 CSL_SrioPortWriteCapt::capture2
Port-Write payload, word 2

Uint32 CSL_SrioPortWriteCapt::capture3
Port-Write payload, word 3

13.3.28 CSL_SrioDirectIO_ConfigXfr

Detailed Description
This structure is used to configure LSU module for Transfer enable.

Field Documentation

Uint16 CSL_SrioDirectIO_ConfigXfr::byteCnt
Number of data bytes to Read/Write - up to 4KB. (Used in conjunction with RapidIO address to create WRSIZE/RDSIZE and WDPTR in RapidIO packet header)

Uint16 CSL_SrioDirectIO_ConfigXfr::doorbellInfo
Doorbell info

Uint16 CSL_SrioDirectIO_ConfigXfr::dstId
RapidIO destination ID field specifying target device

CSL_SrioLongAddress CSL_SrioDirectIO_ConfigXfr::dstNodeAddr
Destination node address

Uint8 CSL_SrioDirectIO_ConfigXfr::hopCount
RapidIO hop count

Uint8 CSL_SrioDirectIO_ConfigXfr::idSize
RapidIO tt field specifying 8 or 16bit Device IDs

Bool CSL_SrioDirectIO_ConfigXfr::intrReq
RapidIO Lsu module interrupt request

Uint8 CSL_SrioDirectIO_ConfigXfr::outPortId
Out port ID

Uint8 CSL_SrioDirectIO_ConfigXfr::pktType
Packet type

Uint8 CSL_SrioDirectIO_ConfigXfr::priority
This field specifies packet priority

Uint32 CSL_SrioDirectIO_ConfigXfr::srcNodeAddr
Source node address

Uint8 CSL_SrioDirectIO_ConfigXfr::xambs
RapidIO xambs field specifying extended address MSB

13.3.29 CSL_SrioSerDesPllCfg

Detailed Description
This structure configures SERDES PLL.

Field Documentation

CSL_SrioSerDesLoopBandwidth CSL_SrioSerDesPllCfg::loopBandwidth
Loop bandwidth

Bool CSL_SrioSerDesPllCfg::pllEnable
Enables the internal PLL of the SERDES

CSL_SrioSerDesPllMply CSL_SrioSerDesPllCfg::pllMplyFactor
PLL multiplication factor

13.3.30 CSL_SrioSerDesRxCfg

Detailed Description
This structure configures the SERDES receiver.

Field Documentation
CSL_SrioSerDesBusWidth CSL_SrioSerDesRxCfg::busWidth
Bus width

Uint8 CSL_SrioSerDesRxCfg::clockDataRecovery
Clock/data recovery configuration

Bool CSL_SrioSerDesRxCfg::enRx
Enable receiver

Uint8 CSL_SrioSerDesRxCfg::equalizer
Configure the adaptive equalizer

Bool CSL_SrioSerDesRxCfg::invertedPolarity
Inverted polarity

CSL_SrioSerDesLos CSL_SrioSerDesRxCfg::los
Loss of signal detection, with selectable thresholds

CSL_SrioSerDesRate CSL_SrioSerDesRxCfg::rate
Operating rate

CSL_SrioSerDesSymAlignment CSL_SrioSerDesRxCfg::symAlign
Enables internal or external symbol alignment.

CSL_SrioSerDesTermination CSL_SrioSerDesRxCfg::termination
Selects input termination options suitable for a variety of AC or DC coupled scenarios

13.3.31 CSL_SrioSerDesTxCfg

Detailed Description
This structure configures the SERDES transmitter.

Field Documentation
CSL_SrioSerDesBusWidth CSL_SrioSerDesTxCfg::busWidth
Bus width

CSL_SrioSerDesCommonMode CSL_SrioSerDesTxCfg::commonMode
Common mode configuration

Bool CSL_SrioSerDesTxCfg::enableFixedPhase

Enable fixed TXBCLKIN[i] phase with TXBCLK [i]

Bool CSL_SrioSerDesTxCfg::enTx

Enable transmitter

Bool CSL_SrioSerDesTxCfg::invertedPolarity

Inverted polarity

Uint8 CSL_SrioSerDesTxCfg::outputDeEmphasis

Output de-emphasis select

CSL_SrioSerDesSwingCfg CSL_SrioSerDesTxCfg::outputSwing

Output swing configuration

CSL_SrioSerDesRate CSL_SrioSerDesTxCfg::rate

Operating rate

13.4 Enumerations

This section lists the enumerations available in the SRIO module.

13.4.1 CSL_SrioHwControlCmd

enum CSL_SrioHwControlCmd
This enum describes the commands used to control the SRIO through CSL_srioHwControl().

Enumeration values:

CSL_SRIO_CMD_PER_ENABLE	Enables/disables the peripheral. **Parameters:** *Bool**
CSL_SRIO_CMD_PLL_CONTROL	Enable/disable the SERDES PLLs. PLL enable macros can be OR'ed to get the value. **Parameters:** *Uint8**
CSL_SRIO_CMD_DOORBELL_INTR_CLEAR	Clears doorbell interrupts. Macros can be OR'ed to get the value. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_LSU_INTR_CLEAR	Clear load/store module interrupts. Macros can be OR'ed to get the value. **Parameters:** *Uint32**
CSL_SRIO_CMD_ERR_RST_INTR_CLEAR	Clears Error, Reset, and Special Event interrupts. Macros can be OR'ed to get the value. **Parameters:** *Uint32**
CSL_SRIO_CMD_DIRECTIO_SRC_NODE_ADDR_SET	Sets 32-bit DSP byte source address. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DIRECTIO_DST_ADDR_MSB_SET	Sets the rapid IO destination MSB address. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DIRECTIO_DST_ADDR_LSB_SET	Sets the rapid IO destination LSB address. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DIRECTIO_XFR_BYTECNT_SET	Number of data bytes to Read/Write - up to 4KB. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DIRECTIO_LSU_XFR_TYPE_SET	Sets 4 MSBs to 4-bit ftype field

	for all packets and 4 LSBs to 4-bit trans field for Packet types 2,5 and 8. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DOORBELL_XFR_SET	Sets RapidIO doorbell info field for type 10 packets and sets the packet type to 10. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_DIRECTIO_LSU_FLOW_MASK_SET	Sets LSU flow masks. Port number is passed as input. Macros can be OR'ed to get the value for argument. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_PORT_COMMAND_SET	Sets the command to be sent in the link-request control symbol. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_SP_ERR_STAT_CLEAR	Clear fields' status of SP_ERR_STAT register. Macros can be OR'ed to get the value to pass the argument. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_LGCL_TRANS_ERR_STAT_CLEAR	Clear status of Logical/Transport layer errors. Macros can be OR'ed to get the value to pass the argument. **Parameters:** *Uint32**
CSL_SRIO_CMD_SP_ERR_DET_STAT_CLEAR	Clears status of port errors interrupts. Macros can be OR'ed to get the value to pass the argument. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_SP_CTL_INDEP_ERR_STAT_CLEAR	Clear the fields status of the SP_CTL_INDEP register. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_CMD_CNTL_SYM_SET	Set control symbols used for packet acknowledgment. **Parameters:** *CSL_SrioCntlSym**
CSL_SRIO_CMD_INTDST_RATE_CNTL	Sets interrupt rate control counter. **Parameters:** *Uint32**

13.4.2 CSL_SrioHwStatusQuery

enum CSL_SrioHwStatusQuery
This enum describes the commands used to get status of various parameters of the SRIO. These values are used in CSL_srioGetHwStatus().

Enumeration values:

CSL_SRIO_QUERY_PID_NUMBER	This query command returns the SRIO Peripheral Identification number. **Parameters:** *CSL_SrioPidNumber**
CSL_SRIO_QUERY_GBL_EN_STAT	Gets global enable status. **Parameters:** *Uint32**
CSL_SRIO_QUERY_BLK_EN_STAT	Gets block enable status for all the blocks. **Parameters:** *CSL_SrioBlkEn**
CSL_SRIO_QUERY_DOORBELL_INTR_STAT	Get doorbell interrupts status. The port number is passed as input. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_LSU_INTR_STAT	Get the LSU interrupts status. **Parameters:** *Uint32**
CSL_SRIO_QUERY_ERR_RST_INTR_STAT	Gets Error, Reset, and Special Event interrupts status. **Parameters:** *Uint32**
CSL_SRIO_QUERY_LSU_INTR_DECODE_STAT	Get status of LSU interrupts decode for DST 0. **Parameters:** *Bool**
CSL_SRIO_QUERY_ERR_INTR_DECODE_STAT	Get Error, Reset, and Special Event interrupts decode status for DST 0. **Parameters:** *Bool**
CSL_SRIO_QUERY_LSU_COMP_CODE_STAT	Gets the status of the pending command of LSU registers for a particular port. **Parameters:** *CSL_SrioLsuCompStat**
CSL_SRIO_QUERY_LSU_BSY_STAT	Gets status of the command registers of LSU module for a particular port. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_DEV_ID_INFO	Gets the type of device (Vendor specific). **Parameters:** *CSL_SrioDevInfo**
CSL_SRIO_QUERY_ASSY_ID_INFO	Gets vendor specific assembly information.

	Parameters: *CSL_SrioAssyInfo**
CSL_SRIO_QUERY_PE_FEATURE	Gets processing element features. **Parameters:** *Uint32**
CSL_SRIO_QUERY_SRC_OPERN_SUPPORT	Get source operations CAR status. **Parameters:** *Uint32**
CSL_SRIO_QUERY_DST_OPERN_SUPPORT	Get destination operations CAR status. **Parameters:** *Uint32**
CSL_SRIO_QUERY_LCL_CFG_BAR	Get local configuration space base addresses. **Parameters:** *CSL_SrioLongAddress**
CSL_SRIO_QUERY_SP_LM_RESP_STAT	Get status of SP_LM_RESP register fields. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_SP_ACKID_STAT	Get status of SP_ACKID_STAT register fields. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_SP_ERR_STAT	Get status of SP_ERR_STAT register fields. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_SP_CTL	Gets SP_CTL register status fields. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_LGCL_TRNS_ERR_STAT	Get the status of logical/transport layer errors. **Parameters:** *Uint32**
CSL_SRIO_QUERY_LGCL_TRNS_ERR_CAPT	Get captured error info of logical/transport layer. **Parameters:** *CSL_SrioLogTrErrInfoCapt**
CSL_SRIO_QUERY_SP_ERR_DET_STAT	Get status of port error detect CSR fields. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_PORT_ERR_CAPT	Get the port error captured information. **Parameters:** *CSL_SrioPortErrCapt**
CSL_SRIO_QUERY_SP_CTL_INDEP	Get port control independent register fields status. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_PW_CAPTURE	Get the port write capture information.

	Parameters: *CSL_SrioPortWriteCapt**
CSL_SRIO_QUERY_ERR_RATE_CNTR_READ	Reads the count of the number of transmission errors that have occurred. **Parameters:** *CSL_SrioPortData**
CSL_SRIO_QUERY_PEAK_ERR_RATE_READ	Reads the peak value of the error rate counter. **Parameters:** *CSL_SrioPortData**

13.4.3 CSL_SrioPortCaptType

enum CSL_SrioPortCaptType
This enum describes type of the captured information at the time of port error.

Enumeration values:

CSL_SRIO_CAPT_TYPE_PKT	Port captured packet data during error
CSL_SRIO_CAPT_TYPE_CNTL_SYM	Port captured control symbols during error
CSL_SRIO_CAPT_TYPE_IMP_SPEC	Port captured implementation specific data during error

13.4.4 CSL_SrioPortNum

enum CSL_SrioPortNum
This enum describes the port number configuration for SRIO.

Enumeration values:

CSL_SRIO_PORT_0	Port number 0
CSL_SRIO_PORT_1	Port number 1
CSL_SRIO_PORT_2	Port number 2
CSL_SRIO_PORT_3	Port number 3

13.4.5 CSL_SrioDiscoveryTimer

enum CSL_SrioDiscoveryTimer
This enum describes the discovery time for the link partner to enter its discovery state.

Enumeration values:

CSL_SRIO_DISCOVERY_TIME_0	Discovery time is 102.4ps (for debug mode only)
CSL_SRIO_DISCOVERY_TIME_1	Discovery time is 0.84ms
CSL_SRIO_DISCOVERY_TIME_2	Discovery time is 0.84ms*2
CSL_SRIO_DISCOVERY_TIME_3	Discovery time is 0.84ms*3
CSL_SRIO_DISCOVERY_TIME_4	Discovery time is 0.84ms*4
CSL_SRIO_DISCOVERY_TIME_5	Discovery time is 0.84ms*5

CSL_SRIO_DISCOVERY_TIME_6	Discovery time is 0.84ms*6
CSL_SRIO_DISCOVERY_TIME_7	Discovery time is 0.84ms*7
CSL_SRIO_DISCOVERY_TIME_8	Discovery time is 0.84ms*8
CSL_SRIO_DISCOVERY_TIME_9	Discovery time is 0.84ms*9
CSL_SRIO_DISCOVERY_TIME_10	Discovery time is 0.84ms*10
CSL_SRIO_DISCOVERY_TIME_11	Discovery time is 0.84ms*11
CSL_SRIO_DISCOVERY_TIME_12	Discovery time is 0.84ms*12
CSL_SRIO_DISCOVERY_TIME_13	Discovery time is 0.84ms*13
CSL_SRIO_DISCOVERY_TIME_14	Discovery time is 0.84ms*14
CSL_SRIO_DISCOVERY_TIME_15	Discovery time is 0.84ms*15

13.4.6 CSL_SrioPwTimer

enum CSL_SrioPwTimer
This enum describes the port write time for request.

Enumeration values:

CSL_SRIO_PW_TIME_0	Port write is sent only once (disabled)
CSL_SRIO_PW_TIME_1	Port write time is 107ms - 214ms
CSL_SRIO_PW_TIME_2	Port write time is 214ms - 321ms
CSL_SRIO_PW_TIME_6	Port write time is 428ms - 535ms
CSL_SRIO_PW_TIME_8	Port write time is 856ms - 963ms
CSL_SRIO_PW_TIME_15	Port write time is 0.82 - 1.64us

13.4.7 CSL_SrioSilenceTimer

enum CSL_SrioSilenceTimer
This enum describes the time values for the port in silent state.

Enumeration values:

CSL_SRIO_SILENCE_TIME_0	Port in silent state for 64ns (debug mode)
CSL_SRIO_SILENCE_TIME_1	Port in silent state for 13.1us*1
CSL_SRIO_SILENCE_TIME_2	Port in silent state for 13.1us*2
CSL_SRIO_SILENCE_TIME_3	Port in silent state for 13.1us*3
CSL_SRIO_SILENCE_TIME_4	Port in silent state for 13.1us*4
CSL_SRIO_SILENCE_TIME_5	Port in silent state for 13.1us*5
CSL_SRIO_SILENCE_TIME_6	Port in silent state for 13.1us*6
CSL_SRIO_SILENCE_TIME_7	Port in silent state for 13.1us*7
CSL_SRIO_SILENCE_TIME_8	Port in silent state for 13.1us*8
CSL_SRIO_SILENCE_TIME_9	Port in silent state for 13.1us*9
CSL_SRIO_SILENCE_TIME_10	Port in silent state for 13.1us*10
CSL_SRIO_SILENCE_TIME_11	Port in silent state for 13.1us*11
CSL_SRIO_SILENCE_TIME_12	Port in silent state for 13.1us*12
CSL_SRIO_SILENCE_TIME_13	Port in silent state for 13.1us*13
CSL_SRIO_SILENCE_TIME_14	Port in silent state for 13.1us*14

CSL_SRIO_SILENCE_TIME_15	Port in silent state for 13.1us*15

13.4.8 CSL_SrioBusTransPriority

enum CSL_SrioBusTransPriority
This enum describes the bus transaction priority values for SRIO.

Enumeration values:

CSL_SRIO_BUS_TRANS_PRIORITY_0	Sets internal bus priority to 0(highest)
CSL_SRIO_BUS_TRANS_PRIORITY_1	Sets internal bus priority to 1
CSL_SRIO_BUS_TRANS_PRIORITY_2	Sets internal bus priority to 2
CSL_SRIO_BUS_TRANS_PRIORITY_3	Sets internal bus priority to 3
CSL_SRIO_BUS_TRANS_PRIORITY_4	Sets internal bus priority to 4
CSL_SRIO_BUS_TRANS_PRIORITY_5	Sets internal bus priority to 5
CSL_SRIO_BUS_TRANS_PRIORITY_6	Sets internal bus priority to 6
CSL_SRIO_BUS_TRANS_PRIORITY_7	Sets internal bus priority to 7

13.4.9 CSL_SrioClkDiv

enum CSL_SrioClkDiv
This enum describes the internal clock prescale values for SRIO.

Enumeration values:

CSL_SRIO_CLK_PRESCALE_0	Sets the internal clock frequency Min 44.7 and Max 89.5
CSL_SRIO_CLK_PRESCALE_1	Sets the internal clock frequency Min 89.5 and Max 179.0
CSL_SRIO_CLK_PRESCALE_2	Sets the internal clock frequency Min 134.2 and Max 268.4
CSL_SRIO_CLK_PRESCALE_3	Sets the internal clock frequency Min 180.0 and Max 360.0
CSL_SRIO_CLK_PRESCALE_4	Sets the internal clock frequency Min 223.7 and Max 447.4
CSL_SRIO_CLK_PRESCALE_5	Sets the internal clock frequency Min 268.4 and Max 536.8
CSL_SRIO_CLK_PRESCALE_6	Sets the internal clock frequency Min 313.2 and Max 626.4
CSL_SRIO_CLK_PRESCALE_7	Sets the internal clock frequency Min 357.9 and Max 715.8
CSL_SRIO_CLK_PRESCALE_8	Sets the internal clock frequency Min 402.6 and Max 805.4
CSL_SRIO_CLK_PRESCALE_9	Sets the internal clock frequency Min 447.4 and Max 894.8
CSL_SRIO_CLK_PRESCALE_10	Sets the internal clock frequency Min 492.1 and Max 984.2
CSL_SRIO_CLK_PRESCALE_11	Sets the internal clock frequency Min 536.9 and Max 1073.8
CSL_SRIO_CLK_PRESCALE_12	Sets the internal clock frequency Min 581.6 and Max 1163.2

CSL_SRIO_CLK_PRESCALE_13	Sets the internal clock frequency Min 626.3 and Max 1252.6
CSL_SRIO_CLK_PRESCALE_14	Sets the internal clock frequency Min 671.1 and Max 1342.2
CSL_SRIO_CLK_PRESCALE_15	Sets the internal clock frequency Min 715.8 and Max 1431.6

13.4.10 CSL_SrioTxPriorityWm

enum CSL_SrioTxPriorityWm
This enum describes required buffer count for packets to be sent across the UDI interface.

Enumeration values:

CSL_SRIO_TX_PRIORITY_WM_0	Transmit credit threshold 1
CSL_SRIO_TX_PRIORITY_WM_1	Transmit credit threshold 2
CSL_SRIO_TX_PRIORITY_WM_2	Transmit credit threshold 3
CSL_SRIO_TX_PRIORITY_WM_3	Transmit credit threshold 4
CSL_SRIO_TX_PRIORITY_WM_4	Transmit credit threshold 5
CSL_SRIO_TX_PRIORITY_WM_5	Transmit credit threshold 6
CSL_SRIO_TX_PRIORITY_WM_6	Transmit credit threshold 7
CSL_SRIO_TX_PRIORITY_WM_7	Transmit credit threshold 8

13.4.11 CSL_SrioAddrSelect

enum CSL_SrioAddrSelect
This enum describes extended addressing control bits.

Enumeration values:

CSL_SRIO_ADDR_SELECT_66BIT	PE supports 66 bit addresses
CSL_SRIO_ADDR_SELECT_50BIT	PE supports 50 bit addresses
CSL_SRIO_ADDR_SELECT_34BIT	PE supports 34 bit addresses (default)

13.4.12 CSL_SrioBufMode

enum CSL_SrioBufMode
This enum describes UDI buffers setup.

Enumeration values:

CSL_SRIO_1X_MODE_PORT	UDI buffers are port based
CSL_SRIO_1X_MODE_PRIORITY	UDI buffers are priority based

13.4.13 CSL_SrioPortWidthOverride

enum CSL_SrioPortWidthOverride
This enum describes the port width override options.

Enumeration values:

CSL_SRIO_PORT_WIDTH_NO_OVERRIDE	No override to the port width
CSL_SRIO_PORT_WIDTH_LANE_0	Force single lane, lane 0
CSL_SRIO_PORT_WIDTH_LANE_2	Force single lane, lane 2

13.4.14 CSL_SrioErrRtBias

enum CSL_SrioErrRtBias
This enum describes the error rate bias values.

Enumeration values:

CSL_SRIO_ERR_RATE_BIAS_0	Error rate counter do not decrement
CSL_SRIO_ERR_RATE_BIAS_1MS	Error rate counter decrements every 1ms
CSL_SRIO_ERR_RATE_BIAS_10MS	Error rate counter decrements every 10ms
CSL_SRIO_ERR_RATE_BIAS_100MS	Error rate counter decrements every 100ms
CSL_SRIO_ERR_RATE_BIAS_1S	Error rate counter decrements every 1s
CSL_SRIO_ERR_RATE_BIAS_10S	Error rate counter decrements every 10s
CSL_SRIO_ERR_RATE_BIAS_100S	Error rate counter decrements every 100s
CSL_SRIO_ERR_RATE_BIAS_1000S	Error rate counter decrements every 1000s
CSL_SRIO_ERR_RATE_BIAS_10000S	Error rate counter decrements every 10000s

13.4.15 CSL_SrioPortLnkTimeout

enum CSL_SrioPortLnkTimeout
This enum describes the port link timeout values.

Enumeration values:

CSL_SRIO_PORT_LNK_TIMEOUT_0	Timer disabled
CSL_SRIO_PORT_LNK_TIMEOUT_1	Timeout value is 205ns
CSL_SRIO_PORT_LNK_TIMEOUT_2	Timeout value is 3.1us
CSL_SRIO_PORT_LNK_TIMEOUT_3	Timeout value is 52.4us
CSL_SRIO_PORT_LNK_TIMEOUT_4	Timeout value is 839.5us
CSL_SRIO_PORT_LNK_TIMEOUT_5	Timeout value is 13.4ms
CSL_SRIO_PORT_LNK_TIMEOUT_6	Timeout value is 215ms
CSL_SRIO_PORT_LNK_TIMEOUT_7	Timeout value is 3.4s

13.4.16 CSL_SrioCompCode

enum CSL_SrioCompCode
This enumeration indicates the status of the pending command.

Enumeration values:

CSL_SRIO_TRANS_NO_ERR	Transaction complete, no errors (Posted/Non-posted)
CSL_SRIO_TRANS_TIMEOUT	Transaction timeout occurred on non-posted transaction
CSL_SRIO_FLOW_CNTL_BLOCKADE	Transaction complete, packet not sent due to flow control blockade (Xoff)
CSL_SRIO_RESP_PKT_ERR	Transaction complete, non-posted response packet (type 8 and 13) contained ERROR status, or response payload length was in error
CSL_SRIO_INV_PROG_ENCODING	Transaction complete, packet not sent due to unsupported transaction type or invalid programming encoding for one or more LSU register fields
CSL_SRIO_DMA_TRANS_ERR	DMA data transfer error
CSL_SRIO_RETRY_DRBL_RESP_RCVD	"Retry" DOORBELL response received, or atomic test-and-swap was not allowed (semaphore in use)
CSL_SRIO_UNAVAILABLE_OUTBOUND_CR	Transaction complete, packet not sent due to unavailable outbound credit at given priority

13.4.17 CSL_SrioErrRtNum

enum CSL_SrioErrRtNum
This enum describes error rate counter threshold values.

Enumeration values:

CSL_SRIO_ERR_RATE_COUNT_2	Only count 2 errors and above
CSL_SRIO_ERR_RATE_COUNT_4	Only count 4 errors and above
CSL_SRIO_ERR_RATE_COUNT_16	Only count 16 errors and above
CSL_SRIO_ERR_RATE_COUNT_NO_LIMIT	No limit in incrementing the error rate count

13.4.18 CSL_SrioSerDesLoopBandwidth

enum CSL_SrioSerDesLoopBandwidth
Enum for SERDES Loop bandwidth.

Enumeration values:

CSL_SRIO_SERDES_LOOP_BANDWIDTH_FREQ_DEP	Frequency dependant loop bandwidth
CSL_SRIO_SERDES_LOOP_BANDWIDTH_LOW	Low loop bandwidth

CSL_SRIO_SERDES_LOOP_BANDWIDTH_HIGH	High loop bandwidth

13.4.19 CSL_SrioSerDesPllMply

enum CSL_SrioSerDesPllMply
Enum for SERDES PLL multiplication factor values.

Enumeration values:

CSL_SRIO_SERDES_PLL_MPLY_BY_4	SERDES PLL multiplication factor 4
CSL_SRIO_SERDES_PLL_MPLY_BY_5	SERDES PLL multiplication factor 5
CSL_SRIO_SERDES_PLL_MPLY_BY_6	SERDES PLL multiplication factor 6
CSL_SRIO_SERDES_PLL_MPLY_BY_8	SERDES PLL multiplication factor 8
CSL_SRIO_SERDES_PLL_MPLY_BY_10	SERDES PLL multiplication factor 10
CSL_SRIO_SERDES_PLL_MPLY_BY_12	SERDES PLL multiplication factor 12
CSL_SRIO_SERDES_PLL_MPLY_BY_12_5	SERDES PLL multiplication factor 12.5
CSL_SRIO_SERDES_PLL_MPLY_BY_15	SERDES PLL multiplication factor 15
CSL_SRIO_SERDES_PLL_MPLY_BY_20	SERDES PLL multiplication factor 20
CSL_SRIO_SERDES_PLL_MPLY_BY_25	SERDES PLL multiplication factor 25
CSL_SRIO_SERDES_PLL_MPLY_BY_50	SERDES PLL multiplication factor 50
CSL_SRIO_SERDES_PLL_MPLY_BY_60	SERDES PLL multiplication factor 60

13.4.20 CSL_SrioSerDesLos

enum CSL_SrioSerDesLos
Enum for SERDES loss of signal detection configuration.

Enumeration values:

CSL_SRIO_SERDES_LOS_DET_DISABLE	Loss of signal detection disabled
CSL_SRIO_SERDES_LOS_DET_HIGH_THRESHOLD	Loss of signal detection threshold in the range 85 to 195mVdfpp.
CSL_SRIO_SERDES_LOS_DET_LOW_THRESHOLD	Loss of signal detection threshold in the range 65 to 175mVdfpp.

13.4.21 CSL_SrioSerDesSymAlignment

enum CSL_SrioSerDesSymAlignment
Enum for SERDES symbol alignment configuration.

Enumeration values:

CSL_SRIO_SERDES_SYM_ALIGN_DISABLE	Symbol alignment disbaled
CSL_SRIO_SERDES_SYM_ALIGN_COMMA	Comma alignment: Symbol alignment will be performed whenever a misaligned comma symbol is received.
CSL_SRIO_SERDES_SYM_ALIGN_JOG	Alignment Jog. The symbol alignment will be adjusted by one bit position

13.4.22 CSL_SrioSerDesTermination

enum CSL_SrioSerDesTermination
Enum for SERDES input termination.

Enumeration values:

CSL_SRIO_SERDES_TERMINATION_VDDT	Input termination is to VDDT
CSL_SRIO_SERDES_TERMINATION_0_8_VDDT	Input termination is to 0.8 VDDT
CSL_SRIO_SERDES_TERMINATION_FLOATING	Input termination is floating

13.4.23 CSL_SrioSerDesRate

enum CSL_SrioSerDesRate
Enum for the SERDES operating rate configuration.

Enumeration values:

CSL_SRIO_SERDES_RATE_FULL	Full rate operation
CSL_SRIO_SERDES_RATE_HALF	Half rate operation
CSL_SRIO_SERDES_RATE_QUARTER	Quarter rate operation

13.4.24 CSL_SrioSerDesBusWidth

enum CSL_SrioSerDesBusWidth
Enum for the SERDES bus width configuration.

Enumeration values:

CSL_SRIO_SERDES_BUS_WIDTH_10_BIT	10 bit bus width
CSL_SRIO_SERDES_BUS_WIDTH_8_BIT	8 bit bus width

13.4.25 CSL_SrioSerDesCommonMode

enum CSL_SrioSerDesCommonMode
Enum for SERDES TX common mode configuration.

Enumeration values:

CSL_SRIO_SERDES_COMMON_MODE_NORMAL	Normal: Common mode not adjusted
CSL_SRIO_SERDES_COMMON_MODE_RAISED	Raised: Common mode raised by 5% of e54

13.4.26 CSL_SrioSerDesSwingCfg

enum CSL_SrioSerDesSwingCfg
Enum for SERDES output swing configuration.

Enumeration values:

CSL_SRIO_SERDES_SWING_AMPLITUDE_125	Output swing amplitude 125
CSL_SRIO_SERDES_SWING_AMPLITUDE_250	Output swing amplitude 250
CSL_SRIO_SERDES_SWING_AMPLITUDE_500	Output swing amplitude 500
CSL_SRIO_SERDES_SWING_AMPLITUDE_625	Output swing amplitude 625
CSL_SRIO_SERDES_SWING_AMPLITUDE_750	Output swing amplitude 750
CSL_SRIO_SERDES_SWING_AMPLITUDE_1000	Output swing amplitude 1000
CSL_SRIO_SERDES_SWING_AMPLITUDE_1125	Output swing amplitude 1125
CSL_SRIO_SERDES_SWING_AMPLITUDE_125 0	Output swing amplitude 1250

13.5 Macros

```
#define CSL_SRIO_DOORBELL_INTR0   (0x00000001)
#define CSL_SRIO_DOORBELL_INTR1  (0x00000002)
#define CSL_SRIO_DOORBELL_INTR2  (0x00000004)
#define CSL_SRIO_DOORBELL_INTR3  (0x00000008)
#define CSL_SRIO_DOORBELL_INTR4  (0x00000010)
#define CSL_SRIO_DOORBELL_INTR5  (0x00000020)
#define CSL_SRIO_DOORBELL_INTR6  (0x00000040)
#define CSL_SRIO_DOORBELL_INTR7  (0x00000080)
#define CSL_SRIO_DOORBELL_INTR8  (0x00000100)
#define CSL_SRIO_DOORBELL_INTR9  (0x00000200)
#define CSL_SRIO_DOORBELL_INTR10 (0x00000400)
#define CSL_SRIO_DOORBELL_INTR11 (0x00000800)
#define CSL_SRIO_DOORBELL_INTR12 (0x00001000)
#define CSL_SRIO_DOORBELL_INTR13 (0x00002000)
#define CSL_SRIO_DOORBELL_INTR14 (0x00004000)
#define CSL_SRIO_DOORBELL_INTR15 (0x00008000)
```

Doorbell interrupts clear macros

```
#define CSL_SRIO_ERR_DEV_RST_INTR   (0x00010000)
#define CSL_SRIO_ERR_PORT3_INTR     (0x00000800)
#define CSL_SRIO_ERR_PORT2_INTR     (0x00000400)
#define CSL_SRIO_ERR_PORT1_INTR     (0x00000200)
#define CSL_SRIO_ERR_PORT0_INTR     (0x00000100)
#define CSL_SRIO_ERR_LGCL_INTR      (0x00000004)
#define CSL_SRIO_ERR_PW_INTR        (0x00000002)
#define CSL_SRIO_ERR_MULTICAST_INTR    (0x00000001)
```

Error, Reset, and Special Event Status Interrupt clear macros

```
#define CSL_SRIO_ERR_IMP_SPECIFIC                  ~(0x80000000)
#define CSL_SRIO_CORRUPT_CNTL_SYM                  ~(0x00400000)
#define CSL_SRIO_CNTL_SYM_UNEXPECTED_ACKID ~(0x00200000)
#define CSL_SRIO_RCVD_PKT_NOT_ACCPT                ~(0x00100000)
#define CSL_SRIO_PKT_UNEXPECTED_ACKID              ~(0x00080000)
#define CSL_SRIO_RCVD_PKT_WITH_BAD_CRC             ~(0x00040000)
#define CSL_SRIO_RCVD_PKT_OVER_276B                ~(0x00020000)
#define CSL_SRIO_NON_OUTSTANDING_ACKID             ~(0x00000020)
#define CSL_SRIO_PROTOCOL_ERROR                    ~(0x00000010)
#define CSL_SRIO_UNSOLICITED_ACK_CNTL_SYM  ~(0x00000002)
#define CSL_SRIO_LINK_TIMEOUT                      ~(0x00000001)
```

Port error detect clear macros

```
#define CSL_SRIO_ERR_IMP_SPECIFIC_ENABLE   (0x80000000)
#define CSL_SRIO_CORRUPT_CNTL_SYM_ENABLE                (0x00400000)
#define CSL_SRIO_CNTL_SYM_UNEXPECTED_ACKID_ENABLE (0x00200000)
#define CSL_SRIO_RCVD_PKT_NOT_ACCPT_ENABLE              (0x00100000)
#define CSL_SRIO_PKT_UNEXPECTED_ACKID_ENABLE            (0x00080000)
#define CSL_SRIO_RCVD_PKT_WITH_BAD_CRC_ENABLE           (0x00040000)
#define CSL_SRIO_RCVD_PKT_OVER_276B_ENABLE              (0x00020000)
#define CSL_SRIO_NON_OUTSTANDING_ACKID_ENABLE           (0x00000020)
#define CSL_SRIO_PROTOCOL_ERROR_ENABLE                  (0x00000010)
#define CSL_SRIO_UNSOLICITED_ACK_CNTL_SYM_ENABLE  (0x00000002)
```

#define CSL_SRIO_LINK_TIMEOUT_ENABLE (0x00000001)

Port error detect enable macros

#define CSL_SRIO_ERR_OUTPUT_PKT_DROP (0x04000000)
#define CSL_SRIO_ERR_OUTPUT_FLD_ENC (0x02000000)
#define CSL_SRIO_ERR_OUTPUT_DEGRD_ENC (0x01000000)
#define CSL_SRIO_ERR_OUTPUT_RETRY_ENC (0x00100000)
#define CSL_SRIO_OUTPUT_ERROR_ENC (0x00020000)
#define CSL_SRIO_INPUT_ERROR_ENC (0x00000200)
#define CSL_SRIO_PORT_WRITE_PND (0x00000010)
#define CSL_SRIO_PORT_ERROR (0x00000004)

Port error Status clear macros

#define CSL_SRIO_IO_ERR_RESP_ENABLE (0x80000000)
#define CSL_SRIO_ILL_TRANS_DECODE_ENABLE (0x08000000)
#define CSL_SRIO_ILL_TRANS_TARGET_ERR_ENABLE (0x04000000)
#define CSL_SRIO_PKT_RESP_TIMEOUT_ENABLE (0x01000000)
#define CSL_SRIO_UNSOLICITED_RESP_ENABLE (0x00800000)
#define CSL_SRIO_UNSUPPORTED_TRANS_ENABLE (0x00400000)

Logical/transport layer error enable

#define CSL_SRIO_IO_ERR_RSPNS ~(0x80000000)
#define CSL_SRIO_ILL_TRANS_DECODE ~(0x08000000)
#define CSL_SRIO_PKT_RSPNS_TIMEOUT ~(0x01000000)
#define CSL_SRIO_UNSOLICITED_RSPNS ~(0x00800000)
#define CSL_SRIO_UNSUPPORTED_TRANS ~(0x00400000)

Logical/transport layer error status clear

#define CSL_SRIO_LSU_INTR0 (0x00000001)
#define CSL_SRIO_LSU_INTR1 (0x00000002)
#define CSL_SRIO_LSU_INTR2 (0x00000004)
#define CSL_SRIO_LSU_INTR3 (0x00000008)
#define CSL_SRIO_LSU_INTR4 (0x00000010)
#define CSL_SRIO_LSU_INTR5 (0x00000020)
#define CSL_SRIO_LSU_INTR6 (0x00000040)
#define CSL_SRIO_LSU_INTR7 (0x00000080)
#define CSL_SRIO_LSU_INTR8 (0x00000100)
#define CSL_SRIO_LSU_INTR9 (0x00000200)
#define CSL_SRIO_LSU_INTR10 (0x00000400)
#define CSL_SRIO_LSU_INTR11 (0x00000800)
#define CSL_SRIO_LSU_INTR12 (0x00001000)
#define CSL_SRIO_LSU_INTR13 (0x00002000)
#define CSL_SRIO_LSU_INTR14 (0x00004000)
#define CSL_SRIO_LSU_INTR15 (0x00008000)
#define CSL_SRIO_LSU_INTR16 (0x00010000)
#define CSL_SRIO_LSU_INTR17 (0x00020000)
#define CSL_SRIO_LSU_INTR18 (0x00040000)
#define CSL_SRIO_LSU_INTR19 (0x00080000)
#define CSL_SRIO_LSU_INTR20 (0x00100000)
#define CSL_SRIO_LSU_INTR21 (0x00200000)
#define CSL_SRIO_LSU_INTR22 (0x00400000)
#define CSL_SRIO_LSU_INTR23 (0x00800000)
#define CSL_SRIO_LSU_INTR24 (0x01000000)
#define CSL_SRIO_LSU_INTR25 (0x02000000)

#define CSL_SRIO_LSU_INTR26 (0x04000000)
#define CSL_SRIO_LSU_INTR27 (0x08000000)
#define CSL_SRIO_LSU_INTR28 (0x10000000)
#define CSL_SRIO_LSU_INTR29 (0x20000000)
#define CSL_SRIO_LSU_INTR30 (0x40000000)
#define CSL_SRIO_LSU_INTR31 (0x80000000)
LSU interrupts clear macros

#define CSL_SRIO_PLL1_ENABLE (0x00000001)
#define CSL_SRIO_PLL2_ENABLE (0x00000002)
#define CSL_SRIO_PLL3_ENABLE (0x00000004)
#define CSL_SRIO_PLL4_ENABLE (0x00000008)
PLL enable macros

#define CSL_SRIO_BLOCKS_MAX 9
Number of blocks

#define CSL_SRIO_FLOW_CONTROL_REG_MAX 16
Number of srio flow control register

#define CSL_SRIO_PORTS_MAX 4
Number of ports

**#define CSL_SRIO_HWSETUP_DEFAULTS **

```
{ \
    CSL_SRIO_PCR_PEREN_RESETVAL, \
    {\
        CSL_SRIO_PER_SET_CNTL_SW_MEM_SLEEP_OVERRIDE_RESETVAL, \
        CSL_SRIO_PER_SET_CNTL_LOOPBACK_RESETVAL, \
        CSL_SRIO_PER_SET_CNTL_BOOT_COMPLETE_RESETVAL, \
        CSL_SRIO_TX_PRIORITY_WM_3, \
        CSL_SRIO_TX_PRIORITY_WM_2, \
        CSL_SRIO_TX_PRIORITY_WM_1, \
        CSL_SRIO_BUS_TRANS_PRIORITY_0, \
        CSL_SRIO_1X_MODE_PRIORITY, \
        CSL_SRIO_CLK_PRESCALE_0, \
        0x0 \
    }, \
    CSL_SRIO_GBL_EN_EN_RESETVAL, \
    {\
        0x0, \
        0x0, \
        0x0, \
        0x0, \
        0x0, \
        0x0, \
        0x0, \
        0x0, \
        0x0 \
    }, \
        CSL_SRIO_DEVICEID_REG1_RESETVAL, \
        CSL_SRIO_DEVICEID_REG2_RESETVAL, \
    { \
```

```
    {0x0000FFFF, 0x0000FFFF, CSL_SRIO_PORT_3, 0x000000FF,
          0x000000FF}, \
    {0x0000FFFF, 0x0000FFFF, CSL_SRIO_PORT_3, 0x000000FF,
          0x000000FF}, \
    {0x0000FFFF, 0x0000FFFF, CSL_SRIO_PORT_3, 0x000000FF,
          0x000000FF}, \
    {0x0000FFFF, 0x0000FFFF, CSL_SRIO_PORT_3, 0x000000FF,
    0x000000FF} \
   }, \
        { \
    { \
    FALSE, \
    CSL_SRIO_SERDES_PLL_MPLY_BY_4, \
    CSL_SRIO_SERDES_LOOP_BANDWIDTH_FREQ_DEP \
          }, \
    { \
    FALSE, \
    CSL_SRIO_SERDES_PLL_MPLY_BY_4, \
    CSL_SRIO_SERDES_LOOP_BANDWIDTH_FREQ_DEP \
          }, \
    { \
    FALSE, \
    CSL_SRIO_SERDES_PLL_MPLY_BY_4, \
    CSL_SRIO_SERDES_LOOP_BANDWIDTH_FREQ_DEP \
          }, \
    { \
    FALSE, \
    CSL_SRIO_SERDES_PLL_MPLY_BY_4, \
    CSL_SRIO_SERDES_LOOP_BANDWIDTH_FREQ_DEP \
          } \
    }, \
  { \
    { \
          FALSE, \
          CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
          CSL_SRIO_SERDES_RATE_FULL, \
          FALSE, \
          CSL_SRIO_SERDES_TERMINATION_VDDT, \
          CSL_SRIO_SERDES_SYM_ALIGN_DISABLE, \
          CSL_SRIO_SERDES_LOS_DET_DISABLE, \
          0x0, \
          0x0 \
          }, \
    { \
          FALSE, \
          CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
          CSL_SRIO_SERDES_RATE_FULL, \
          FALSE, \
          CSL_SRIO_SERDES_TERMINATION_VDDT, \
          CSL_SRIO_SERDES_SYM_ALIGN_DISABLE, \
          CSL_SRIO_SERDES_LOS_DET_DISABLE, \
          0x0, \
          0x0 \
          }, \
    { \
          FALSE, \
```

```
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
            FALSE, \
            CSL_SRIO_SERDES_TERMINATION_VDDT, \
            CSL_SRIO_SERDES_SYM_ALIGN_DISABLE, \
            CSL_SRIO_SERDES_LOS_DET_DISABLE, \
            0x0, \
            0x0 \
            }, \
      { \
            FALSE, \
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
            FALSE, \
            CSL_SRIO_SERDES_TERMINATION_VDDT, \
            CSL_SRIO_SERDES_SYM_ALIGN_DISABLE, \
            CSL_SRIO_SERDES_LOS_DET_DISABLE, \
            0x0, \
            0x0 \
            } \
      }, \
    { \
      { \
            FALSE, \
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
            FALSE, \
            CSL_SRIO_SERDES_COMMON_MODE_NORMAL, \
            CSL_SRIO_SERDES_SWING_AMPLITUDE_125, \
            0x0, \
            FALSE \
            }, \
      { \
            FALSE, \
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
            FALSE, \
            CSL_SRIO_SERDES_COMMON_MODE_NORMAL, \
            CSL_SRIO_SERDES_SWING_AMPLITUDE_125, \
            0x0, \
            FALSE \
            }, \
      { \
            FALSE, \
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
            FALSE, \
            CSL_SRIO_SERDES_COMMON_MODE_NORMAL, \
            CSL_SRIO_SERDES_SWING_AMPLITUDE_125, \
            0x0, \
            FALSE \
            }, \
      { \
            FALSE, \
            CSL_SRIO_SERDES_BUS_WIDTH_10_BIT, \
            CSL_SRIO_SERDES_RATE_FULL, \
```

```
            FALSE, \
            CSL_SRIO_SERDES_COMMON_MODE_NORMAL, \
            CSL_SRIO_SERDES_SWING_AMPLITUDE_125, \
            0x0, \
            FALSE \
            } \
    }, \
{0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1,
\
0x1, 0x1 }, \
     {0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, \
      0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000 }, \

(CSL_SrioAddrSelect)CSL_SRIO_PE_LL_CTL_EXTENDED_ADDRESSING_CONTROL_RESE
TVAL, \
     {\
        CSL_SRIO_BASE_ID_BASE_DEVICEID_RESETVAL, \
        CSL_SRIO_BASE_ID_LARGE_BASE_DEVICEID_RESETVAL, \
        CSL_SRIO_HOST_BASE_ID_LOCK_HOST_BASE_DEVICEID_RESETVAL \
    }, \
    CSL_SRIO_COMP_TAG_COMPONENT_TAG_RESETVAL, \
    {\
        CSL_SRIO_SP_LT_CTL_TIMEOUT_VALUE_RESETVAL, \
        CSL_SRIO_SP_RT_CTL_TIMEOUT_VALUE_RESETVAL, \
        0x0, \
        0x0 \
    }, \
    { \
        {\
            FALSE, \
            FALSE, \
            FALSE, \

(CSL_SrioPortWidthOverride)CSL_SRIO_SP_CTL_PORT_WIDTH_OVERRIDE_RESETVAL
, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE \
        },\
        {\
            FALSE, \
            FALSE, \
            FALSE, \

(CSL_SrioPortWidthOverride)CSL_SRIO_SP_CTL_PORT_WIDTH_OVERRIDE_RESETVAL
, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE \
        }, \
        {\
```

```
            FALSE, \
            FALSE, \
            FALSE, \

(CSL_SrioPortWidthOverride)CSL_SRIO_SP_CTL_PORT_WIDTH_OVERRIDE_RESETVAL
, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE \
        }, \
        {\
            FALSE, \
            FALSE, \
            FALSE, \

(CSL_SrioPortWidthOverride)CSL_SRIO_SP_CTL_PORT_WIDTH_OVERRIDE_RESETVAL
, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE, \
            FALSE \
        } \
    }, \
    CSL_SRIO_ERR_EN_RESETVAL, \
    {\
        {\
            CSL_SRIO_SP_RATE_EN_RESETVAL, \
(CSL_SrioErrRtBias)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_BIAS_RESETVAL, \

(CSL_SrioErrRtNum)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_RECOVERY_RESETVAL, \

CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_FAILED_THRESHOLD_RESETVAL, \
CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_DEGRADED_THRES_RESETVAL \
         }, \
         {\
             CSL_SRIO_SP_RATE_EN_RESETVAL, \

(CSL_SrioErrRtBias)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_BIAS_RESETVAL, \

(CSL_SrioErrRtNum)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_RECOVERY_RESETVAL, \

   CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_FAILED_THRESHOLD_RESETVAL, \
   CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_DEGRADED_THRES_RESETVAL \
         }, \
        {\
            CSL_SRIO_SP_RATE_EN_RESETVAL, \

(CSL_SrioErrRtBias)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_BIAS_RESETVAL, \

(CSL_SrioErrRtNum)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_RECOVERY_RESETVAL, \

CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_FAILED_THRESHOLD_RESETVAL, \
```

```
              CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_DEGRADED_THRES_RESETVAL
\
          }, \
          {\
              CSL_SRIO_SP_RATE_EN_RESETVAL, \
(CSL_SrioErrRtBias)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_BIAS_RESETVAL, \

(CSL_SrioErrRtNum)CSL_SRIO_SP_ERR_RATE_ERROR_RATE_RECOVERY_RESETVAL, \

CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_FAILED_THRESHOLD_RESETVAL, \
              CSL_SRIO_SP_ERR_THRESH_ERROR_RATE_DEGRADED_THRES_RESETVAL
\
          } \
      }, \

(CSL_SrioDiscoveryTimer)CSL_SRIO_SP_IP_DISCOVERY_TIMER_DISCOVERY_TIMER_
RESETVAL, \
    CSL_SRIO_SP_IP_MODE_RESETVAL, \
    CSL_SRIO_IP_PRESCAL_RESETVAL, \
    (CSL_SrioPwTimer)CSL_SRIO_SP_IP_DISCOVERY_TIMER_PW_TIMER_RESETVAL,
\
     { \

(CSL_SrioSilenceTimer)CSL_SRIO_SP_SILENCE_TIMER_SILENCE_TIMER_RESETVAL,
\

(CSL_SrioSilenceTimer)CSL_SRIO_SP_SILENCE_TIMER_SILENCE_TIMER_RESETVAL,
\

(CSL_SrioSilenceTimer)CSL_SRIO_SP_SILENCE_TIMER_SILENCE_TIMER_RESETVAL,
\

(CSL_SrioSilenceTimer)CSL_SRIO_SP_SILENCE_TIMER_SILENCE_TIMER_RESETVAL
\
     }, \
     { \
         CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
         CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
         CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
         CSL_SRIO_SP_CTL_INDEP_RESETVAL  \
         }\
 }
```

Default hardware setup parameters

**#define CSL_SRIO_CONFIG_DEFAULTS **

```
{ \
     CSL_SRIO_PCR_RESETVAL, \
     CSL_SRIO_PER_SET_CNTL_RESETVAL, \
     CSL_SRIO_GBL_EN_RESETVAL, \
     {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, \
     CSL_SRIO_DEVICEID_REG1_RESETVAL, \
     CSL_SRIO_DEVICEID_REG2_16BNODEID_RESETVAL, \
     {\
         {0xFFFFFFFF, 0x0003FFFF},\
         {0xFFFFFFFF, 0x0003FFFF}, \
```

```
        {0xFFFFFFFF, 0x0003FFFF}, \
        {0xFFFFFFFF, 0x0003FFFF} \
    }, \
    {0x00000000, 0x00000000, 0x00000000, 0x00000000}, \
    {0x00000000, 0x00000000, 0x00000000, 0x00000000}, \
    {0x00000000, 0x00000000, 0x00000000, 0x00000000}, \
    {0x00000000, 0x00000000, 0x00000000, 0x00000000}, \
    CSL_SRIO_LSU_ICCR_RESETVAL, \
    CSL_SRIO_ERR_RST_EVNT_ICCR_RESETVAL, \
    CSL_SRIO_INTDST_RATE_CNTL_RESETVAL, \
    {\
        {\
            CSL_SRIO_LSU_REG0_RESETVAL, \
            CSL_SRIO_LSU_REG1_RESETVAL, \
            CSL_SRIO_LSU_REG2_RESETVAL, \
            CSL_SRIO_LSU_REG3_RESETVAL, \
            CSL_SRIO_LSU_REG4_RESETVAL, \
            CSL_SRIO_LSU_FLOW_MASKS_RESETVAL \
        }, \
        {\
            CSL_SRIO_LSU_REG0_RESETVAL, \
            CSL_SRIO_LSU_REG1_RESETVAL, \
            CSL_SRIO_LSU_REG2_RESETVAL, \
            CSL_SRIO_LSU_REG3_RESETVAL, \
            CSL_SRIO_LSU_REG4_RESETVAL, \
            CSL_SRIO_LSU_FLOW_MASKS_RESETVAL \
        }, \
        {\
           CSL_SRIO_LSU_REG0_RESETVAL, \
           CSL_SRIO_LSU_REG1_RESETVAL, \
           CSL_SRIO_LSU_REG2_RESETVAL, \
           CSL_SRIO_LSU_REG3_RESETVAL, \
           CSL_SRIO_LSU_REG4_RESETVAL, \
           CSL_SRIO_LSU_FLOW_MASKS_RESETVAL \
       }, \
       {\
           CSL_SRIO_LSU_REG0_RESETVAL, \
           CSL_SRIO_LSU_REG1_RESETVAL, \
           CSL_SRIO_LSU_REG2_RESETVAL, \
           CSL_SRIO_LSU_REG3_RESETVAL, \
           CSL_SRIO_LSU_REG4_RESETVAL, \
           CSL_SRIO_LSU_FLOW_MASKS_RESETVAL \
       } \
   }, \
   {\
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
       CSL_SRIO_FLOW_CNTL_RESETVAL, \
```

```
        CSL_SRIO_FLOW_CNTL_RESETVAL, \
        CSL_SRIO_FLOW_CNTL_RESETVAL, \
        CSL_SRIO_FLOW_CNTL_RESETVAL, \
        CSL_SRIO_FLOW_CNTL_RESETVAL, \
        CSL_SRIO_FLOW_CNTL_RESETVAL \
    }, \
    CSL_SRIO_PE_LL_CTL_RESETVAL, \
    CSL_SRIO_BASE_ID_RESETVAL, \
    CSL_SRIO_HOST_BASE_ID_LOCK_RESETVAL, \
    CSL_SRIO_COMP_TAG_RESETVAL, \
    CSL_SRIO_SP_LT_CTL_RESETVAL, \
    CSL_SRIO_SP_RT_CTL_RESETVAL, \
    CSL_SRIO_SP_GEN_CTL_RESETVAL, \
    {\
        {\
            CSL_SRIO_SP_LM_REQ_RESETVAL, \
            CSL_SRIO_SP_ACKID_STAT_RESETVAL, \
            CSL_SRIO_SP_ERR_STAT_RESETVAL, \
            CSL_SRIO_SP_CTL_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_LM_REQ_RESETVAL, \
            CSL_SRIO_SP_ACKID_STAT_RESETVAL, \
            CSL_SRIO_SP_ERR_STAT_RESETVAL, \
            CSL_SRIO_SP_CTL_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_LM_REQ_RESETVAL, \
            CSL_SRIO_SP_ACKID_STAT_RESETVAL, \
            CSL_SRIO_SP_ERR_STAT_RESETVAL, \
            CSL_SRIO_SP_CTL_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_LM_REQ_RESETVAL, \
            CSL_SRIO_SP_ACKID_STAT_RESETVAL, \
            CSL_SRIO_SP_ERR_STAT_RESETVAL, \
            CSL_SRIO_SP_CTL_RESETVAL \
        } \
    }, \
    CSL_SRIO_ERR_DET_RESETVAL, \
    CSL_SRIO_ERR_EN_RESETVAL, \
    CSL_SRIO_PW_TGT_ID_RESETVAL, \
    {\
        {\
            CSL_SRIO_SP_ERR_DET_RESETVAL, \
            CSL_SRIO_SP_RATE_EN_RESETVAL, \
            CSL_SRIO_SP_ERR_RATE_RESETVAL, \
            CSL_SRIO_SP_ERR_THRESH_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_ERR_DET_RESETVAL, \
            CSL_SRIO_SP_RATE_EN_RESETVAL, \
            CSL_SRIO_SP_ERR_RATE_RESETVAL, \
            CSL_SRIO_SP_ERR_THRESH_RESETVAL \
        }, \
        {\
```

```
            CSL_SRIO_SP_ERR_DET_RESETVAL, \
            CSL_SRIO_SP_RATE_EN_RESETVAL, \
            CSL_SRIO_SP_ERR_RATE_RESETVAL, \
            CSL_SRIO_SP_ERR_THRESH_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_ERR_DET_RESETVAL, \
            CSL_SRIO_SP_RATE_EN_RESETVAL, \
            CSL_SRIO_SP_ERR_RATE_RESETVAL, \
            CSL_SRIO_SP_ERR_THRESH_RESETVAL \
        } \
    }, \
    CSL_SRIO_SP_IP_DISCOVERY_TIMER_RESETVAL, \
    CSL_SRIO_SP_IP_MODE_RESETVAL, \
    CSL_SRIO_IP_PRESCAL_RESETVAL, \
    {\
        {\
            CSL_SRIO_SP_RST_OPT_RESETVAL, \
            CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
            CSL_SRIO_SP_SILENCE_TIMER_RESETVAL, \
            CSL_SRIO_SP_MULT_EVNT_CS_RESETVAL, \
            CSL_SRIO_SP_CS_TX_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_RST_OPT_RESETVAL, \
            CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
            CSL_SRIO_SP_SILENCE_TIMER_RESETVAL, \
            CSL_SRIO_SP_MULT_EVNT_CS_RESETVAL, \
            CSL_SRIO_SP_CS_TX_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_RST_OPT_RESETVAL, \
            CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
            CSL_SRIO_SP_SILENCE_TIMER_RESETVAL, \
            CSL_SRIO_SP_MULT_EVNT_CS_RESETVAL, \
            CSL_SRIO_SP_CS_TX_RESETVAL \
        }, \
        {\
            CSL_SRIO_SP_RST_OPT_RESETVAL, \
            CSL_SRIO_SP_CTL_INDEP_RESETVAL, \
            CSL_SRIO_SP_SILENCE_TIMER_RESETVAL, \
            CSL_SRIO_SP_MULT_EVNT_CS_RESETVAL, \
            CSL_SRIO_SP_CS_TX_RESETVAL \
        } \
    }\
}
```

Default values for config structure

13.6 Typedefs

typedef CSL_SrioObj * CSL_SrioHandle
This data type is used to return the handle to the CSL of the SRIO.

Chapter 14
TCP2 Module

Topics

14.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within TCP2 module.

The Turbo Decoder Coprocessor (TCP) is a programmable peripheral for decoding of IS2000/3GPP turbo codes. The TCP is controlled via memory mapped control registers and data buffers. The coprocessor operates either as a complete turbo decoder including the iterative structure (standalone processing mode), or it can operate as a single MAP (Maximum A Posterior) decoder (shared processing mode). In the standalone processing mode, the inputs into the TCP are channel soft decisions for systematic and parity bits, and the outputs are hard decisions. In the shared processing mode, the inputs are channel soft decisions for systematic and parity bits and apriori information for systematic bits, and the outputs are extrinsic information for systematic bits.

TCP is enhanced Turbo code system and will be able to support 44 384 Kbps data channels running at 333 Mhz.

The TCP2 supports:

- Parallel concatenated convolutional turbo decoding using the MAP algorithm
- All turbo code rates greater than or equal to 1/5
- 3GPP and CDMA2000 turbo encoder trellis
- 3GPP and CDMA2000 block sizes in standalone mode
- Larger block sizes in shared processing mode
- Both max log MAP and log MAP decoding
- Sliding windows algorithm with variable reliability and prolog lengths
- The prolog reduction algorithm
- Execution of a minimum and maximum number of iterations
- The SNR stopping criteria algorithm
- The CRC stopping criteria algorithm

14.2 Functions

This section lists the functions available in the TCP2 module CSL.

14.2.1 TCP2_setParams

void TCP2_setParams ([TCP2_Params](#) ***restrict** ***configParams***,
[TCP2_ConfigIc](#) ***restrict** ***configIc***
)

Description
This function sets up the TCP input configuration parameters in the TCP2_ConfigIc structure. The configuration values are passed in the configParams input argument.

Arguments

```
configParams    Pointer to the structure holding the TCP
                configuration parameters.

configIc        Pointer to the TCP2_ConfigIc structure to be filled.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configIc argument passed.

Example

```
TCP2_ConfigIc     configIc;
Uint32            cnt;
TCP2_BaseParams   configBase;
TCP2_Params       configParams;
Uint32            frameLen = 40;

// Assign the configuration parameters
configBase.frameLen      = frameLen;
configBase.inputSign     = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag       = 1;
configBase.maxIter       = 8;
configBase.maxStarEn     = TRUE;
configBase.standard      = TCP2_STANDARD_3GPP;
configBase.crcLen        = 0;
configBase.crcPoly       = 0;
configBase.minIter       = 1;
configBase.numCrcPass    = 1;
configBase.outParmFlag   = 0;
```

```
configBase.outputOrder  = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn  = FALSE;
configBase.prologSize   = 24;
configBase.rate         = TCP2_RATE_1_3;
configBase.snr          = 0;

for (cnt = 0; cnt < 16; cnt++)
    configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams(&configBase, &configParams);

// Generate the configuration register values
TCP2_setParams(&configParams, &configIc);
...
```

14.2.2 TCP2_tailConfig

void TCP2_tailConfig	**(**	TCP2_Standard	***standard*,**
		TCP2_Mode	***mode*,**
		TCP2_Map	***map*,**
		TCP2_Rate	***rate*,**
		TCP2_TailData ***restrict**	***tailData*,**
		TCP2_ConfigIc ***restrict**	***configIc***
	)		

Description
This function generates the input control values IC6-IC11 based on the processing to be performed by the TCP. These values consist of the tail data following the systematics and parities data. This function calls specific tail generation functions depending on the standard followed.

Arguments

```
standard        3G standard to be decoded.

mode            TCP processing mode (SA or SP)

map             TCP shared processing MAP

rate            Code rate of the TCP

tailData        Pointer to the tail data

configIc        Pointer to the TCP2_ConfigIc structure to be filled.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configIc argument passed.

Example

```
TCP2_ConfigIc    configIc;
TCP2_BaseParams   configBase;
TCP2_Params       configParams;
Uint32            frameLen = 40;
Uint32            cnt;

TCP2_TailData tailData []= {  0x2f,
                              0x31,
                              0x30,
                              0x20,
                              0x32,
                              0x27,
                              0x30,
                              0x0d,
                              0x10,
                              0x3f,
                              0x18,
                              0x3b };

  TCP2_UserData *xabData = &tailData [frameLen];

// Assign the configuration parameters
configBase.mode         = TCP2_MODE_SA;
configBase.frameLen     = frameLen;
configBase.inputSign    = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag      = 1;
configBase.maxIter      = 8;
configBase.maxStarEn    = TRUE;
configBase.standard     = TCP2_STANDARD_3GPP;
configBase.crcLen       = 0;
configBase.crcPoly      = 0;
configBase.minIter      = 1;
configBase.numCrcPass   = 1;
configBase.outParmFlag  = 0;
configBase.outputOrder  = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn  = FALSE;
configBase.prologSize   = 24;
configBase.rate         = TCP2_RATE_1_3;
configBase.snr          = 0;

for (cnt = 0; cnt < 16; cnt++)
      configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams (&configBase, &configParams);

// Generate the configuration register values
TCP2_setParams (&configParams, &configIc);
```

```
TCP2_tailConfig ( configParams.standard,
                  configParams.mode,
                  configParams.map,
                  configParams.rate,
                  xabData, &configIc);
```

14.2.3 TCP2_genIc

void TCP2_genIc	**(**	**TCP2_Params *restrict**	***configParams*,**
		TCP2_TailData *restrict	***tailData*,**
		TCP2_ConfigIc *restrict	***configIc***
	)		

Description
This function sets up the TCP input configuration parameters in the TCP2_ConfigIc structure. The configuration values are passed in the configParams input argument.

Arguments

configParams	Pointer to the structure holding the TCP configuration parameters.
tailData	Tail data
configIc	Pointer to the TCP2_ConfigIc structure to be filled.

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configIc argument passed.

Example

```
TCP2_TailData tailData []= {  0x2f,
                        0x31,
                        0x30,
                        0x20,
                        0x32,
                        0x27,
                        0x30,
                        0x0d,
                        0x10,
                        0x3f,
                        0x18,
```

```
                            0x3b };

        TCP2_ConfigIc          configIc;
        TCP2_BaseParams        configBase;
        TCP2_Params            configParams;
        TCP2_TailData          *xabData;
        Uint32                 frameLen = 40;
        Uint32                 cnt;

        xabData = &tailData [frameLen];

        // Assign the configuration parameters
        configBase.mode              = TCP2_MODE_SA;
        configBase.frameLen          = frameLen;
        configBase.inputSign         = TCP2_INPUT_SIGN_POSITIVE;
        configBase.intFlag           = 1;
        configBase.maxIter           = 8;
        configBase.maxStarEn         = TRUE;
        configBase.standard          = TCP2_STANDARD_3GPP;
        configBase.crcLen            = 0;
        configBase.crcPoly           = 0;
        configBase.minIter           = 1;
        configBase.numCrcPass        = 1;
        configBase.outParmFlag       = 0;
        configBase.outputOrder       = TCP2_OUT_ORDER_0_31;
        configBase.prologRedEn       = FALSE;
        configBase.prologSize        = 24;
        configBase.rate              = TCP2_RATE_1_3;
        configBase.snr               = 0;

        for (cnt = 0; cnt < 16; cnt++)
              configBase.extrScaling [cnt] = 32;

        // Setup the TCP configuration registers parameters
        TCP2_genParams (&configBase, &configParams);

        // Generate the configuration register values
        TCP2_genIc (&configParams, xabData, &configIc);
```

14.2.4 TCP2_genParams

Uint32 TCP2_genParams	**(**	**TCP2_BaseParams *restrict**	***configBase*,**
		TCP2_Params *restrict	***configParams***
	)		

Description

This function copies the basic parameters, to the configParams parameters structure. For shared processing mode this function copies the configuration parameters for the first/middle sub-frame and the last sub frame. Hence, for this mode the function expects the configParams to be a pointer to an array of two TCP2_Params structure.

Arguments

```
configBase       Pointer to the TCP2_BaseParams structure

configParams     Pointer to the TCP configuration parameters
                 structure.
```

Return Value

`Uint32`

- The number of sub frames for shared processing mode.

Pre Condition

configBase is populated with all the configuration parameters

Post Condition

None

Modifies

The configParams argument passed.

Example

```
TCP2_BaseParams  configBase;
TCP2_Params      configParams;
Uint32           frameLen = 40, cnt;


// Assign the configuration parameters
configBase.frameLen          = frameLen;
configBase.inputSign         = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag           = 1;
configBase.maxIter           = 8;
configBase.maxStarEn         = TRUE;
configBase.standard          = TCP2_STANDARD_3GPP;
configBase.crcLen            = 0;
configBase.crcPoly           = 0;
configBase.minIter           = 1;
configBase.numCrcPass        = 1;
configBase.outParmFlag       = 0;
configBase.outputOrder       = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn       = FALSE;
configBase.prologSize        = 24;
configBase.rate              = TCP2_RATE_1_3;
configBase.snr               = 0;

for (cnt = 0; cnt < 16; cnt++)
    configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams(&configBase, &configParams);
...
```

14.2.5 TCP2_calcSubBlocksSA

void TCP2_calcSubBlocksSA (TCP2_Params * *configParams*)

Description
This function calculates the number of sub blocks for the TCP standalone processing. The reliability length is also calculated.The configParams structure is populated with the calculated parameters.

Arguments

```
configParams      Pointer to the structure holding the TCP
                  configuration parameters.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configParams argument passed.

Example

```
TCP2_BaseParams         configBase;
TCP2_Params             configParams;
Uint32                  frameLen = 40;
Uint8                   cnt;

// Assign the configuration parameters
configBase.mode               = TCP2_MODE_SA;
configBase.frameLen           = frameLen;
configBase.inputSign          = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag            = 1;
configBase.maxIter            = 8;
configBase.maxStarEn          = TRUE;
configBase.standard           = TCP2_STANDARD_3GPP;
configBase.crcLen             = 0;
configBase.crcPoly            = 0;
configBase.minIter            = 1;
configBase.numCrcPass         = 1;
configBase.outParmFlag        = 0;
configBase.outputOrder        = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn        = FALSE;
configBase.prologSize         = 24;
configBase.rate               = TCP2_RATE_1_3;
configBase.snr                = 0;

for (cnt = 0; cnt < 16; cnt++)
          configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams (&configBase, &configParams);
```

```
TCP2_calcSubBlocksSA (&configParams);
```

14.2.6 TCP2_calcSubBlocksSP

Uint32 TCP2_calcSubBlocksSP (TCP2_Params * *configParams*)

Description
This function calculates the number of sub blocks for the TCP shared processing. The reliability length is also calculated and the configParams structure is populated. These parameters are caculated for the first/middle sub-frame and the last sub frame. The function expects the configParams to be a pointer to an array of two TCP2_Params structure.

Arguments

```
configParams      Pointer to the structure holding the TCP
                  configuration parameters.
```

Return Value
`Uint32`

- Number of sub frames the frame is divided into

Pre Condition
None

Post Condition
None

Modifies
The configParams argument passed.

Example

```
  TCP2_BaseParams       configBase;
TCP2_Params             configParams;
Uint32                  frameLen = 40;
Uint8                   cnt;
Uint32                  numSubFrames;

// Assign the configuration parameters
configBase.mode               = TCP2_MODE_SA;
configBase.frameLen           = frameLen;
configBase.inputSign          = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag            = 1;
configBase.maxIter            = 8;
configBase.maxStarEn          = TRUE;
configBase.standard           = TCP2_STANDARD_3GPP;
configBase.crcLen             = 0;
configBase.crcPoly            = 0;
configBase.minIter            = 1;
configBase.numCrcPass         = 1;
configBase.outParmFlag        = 0;
configBase.outputOrder        = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn        = FALSE;
```

```
configBase.prologSize         = 24;
configBase.rate               = TCP2_RATE_1_3;
configBase.snr                = 0;

for (cnt = 0; cnt < 16; cnt++)
        configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams (&configBase, &configParams);
numSubFrames = TCP2_calcSubBlocksSP (&configParams);
...
```

14.2.7 TCP2_tailConfig3GPP

void TCP2_tailConfig3GPP	**(**	TCP2_Mode	***mode*,**
		TCP2_Map	***map*,**
		TCP2_Rate	***rate*,**
		TCP2_TailData ***restrict**	***tailData*,**
		TCP2_ConfigIc ***restrict**	***configIc***
	)		

Description
This function generates the input control values IC6-IC11 for 3GPP channels. These values consist of the tail data following the systematics and parities data. This function is called from the generic TCP2_tailConfig function.

Arguments

```
mode        TCP processing mode (SA or SP)

map         TCP shared processing MAP

rate        TCP data code rate

tailData    Pointer to the tail data

configIc    Pointer to the IC values structure
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configIc argument passed.

Example

```
            TCP2_ConfigIc           configIc;
TCP2_BaseParams         configBase;
TCP2_Params             configParams;
Uint32                  frameLen = 40;
Uint16                  cnt;
TCP2_TailData tailData []= {  0x2f,
                              0x31,
                              0x30,
                              0x20,
                              0x32,
                              0x27,
                              0x30,
                              0x0d,
                              0x10,
                              0x3f,
                              0x18,
                              0x3b  };
TCP2_TailData *xabData = &tailData[frameLen];

// Assign the configuration parameters
configBase.mode         = TCP2_MODE_SA;
configBase.frameLen     = frameLen;
configBase.inputSign    = TCP2_INPUT_SIGN_POSITIVE;
configBase.intFlag      = 1;
configBase.maxIter      = 8;
configBase.maxStarEn    = TRUE;
configBase.standard     = TCP2_STANDARD_3GPP;
configBase.crcLen       = 0;
configBase.crcPoly      = 0;
configBase.minIter      = 1;
configBase.numCrcPass   = 1;
configBase.outParmFlag  = 0;
configBase.outputOrder  = TCP2_OUT_ORDER_0_31;
configBase.prologRedEn  = FALSE;
configBase.prologSize   = 24;
configBase.rate         = TCP2_RATE_1_3;
configBase.snr          = 0;

for (cnt = 0; cnt < 16; cnt++)
   configBase.extrScaling [cnt] = 32;

// Setup the TCP configuration registers parameters
TCP2_genParams (&configBase, &configParams);

// Generate the configuration register values
TCP2_setParams (&configParams, &configIc);

TCP2_tailConfig3GPP(configParams.mode,
                  configParams.map,
                  configParams.rate,
                  xabData, &configIc);
```

14.2.8 TCP2_tailConfigIs2000

void TCP2_tailConfigIs2000	(	TCP2_Mode	***mode***,
		TCP2_Map	***map***,
		TCP2_Rate	***rate***,
		TCP2_TailData ***restrict**	***tailData***,
		TCP2_ConfigIc ***restrict**	***configIc***
	)		

Description
This function generates the input control values IC6-IC11 for IS2000 channels. These values consist of the tail data following the systematics and parities data. This function is called from the generic TCP2_tailConfig function.

Arguments

```
mode        TCP processing mode (SA or SP)

map         TCP shared processing MAP

rate        TCP data code rate

tailData    Pointer to the tail data

configIc    Pointer to the IC values structure
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The configIc argument passed.

Example

```
      TCP2_ConfigIc            configIc;
TCP2_BaseParams         configBase;
TCP2_Params             configParams;
Uint32                  frameLen = 40;
Uint16                  cnt;
TCP2_TailData tailData []= {  0x2f,
                              0x31,
                              0x30,
                              0x20,
                              0x32,
                              0x27,
                              0x30,
                              0x0d,
                              0x10,
```

```
                              0x3f,
                              0x18,
                              0x3b };
 TCP2_TailData *xabData = &tailData[frameLen];

// Assign the configuration parameters
 configBase.mode          = TCP2_MODE_SA;
 configBase.frameLen      = frameLen;
 configBase.inputSign     = TCP2_INPUT_SIGN_POSITIVE;
 configBase.intFlag       = 1;
 configBase.maxIter       = 8;
 configBase.maxStarEn     = TRUE;
 configBase.standard      = TCP2_STANDARD_3GPP;
 configBase.crcLen        = 0;
 configBase.crcPoly       = 0;
 configBase.minIter       = 1;
 configBase.numCrcPass    = 1;
 configBase.outParmFlag   = 0;
 configBase.outputOrder   = TCP2_OUT_ORDER_0_31;
 configBase.prologRedEn   = FALSE;
 configBase.prologSize    = 24;
 configBase.rate          = TCP2_RATE_1_3;
 configBase.snr           = 0;

 for (cnt = 0; cnt < 16; cnt++)
    configBase.extrScaling [cnt] = 32;

 // Setup the TCP configuration registers parameters
 TCP2_genParams (&configBase, &configParams);

 // Generate the configuration register values
 TCP2_setParams (&configParams, &configIc);

 TCP2_tailConfigIs2000 ( configParams.mode,
                         configParams.map,
                         configParams.rate,
                         xabData, &configIc);
```

14.2.9 TCP2_deinterleaveExt

void TCP2_deinterleaveExt	**(**	**TCP2_ExtrinsicData ***	***aprioriMap1*,**
		const TCP2_ExtrinsicData*	***extrinsicsMap2*,**
		const Uint16 *	***interleaverTable*,**
		Uint32	***numExt***
	)		

Description
This function de-interleaves the MAP2 extrinsics data to generate apriori data for the MAP1 decode. This function is for use in performing shared processing.

Arguments

`aprioriMap1`	`Apriori data for MAP1 decode`
`extrinsicsMap2`	`Extrinsics data`
`interleaverTable`	`Interleaver data table`
`numExt`	`Number of Extrinsics`

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The aprioriMap1 argument passed is modified to contain the deinterleaved data.

Example

```
extern TCP2_ExtrinsicData     *aprioriMap1;
extern TCP2_ExtrinsicData     *extrinsicsMap2;
extern Uint16                 *interleaverTable;
Uint32 numExt = 20800;

<...MAP 2 decode...>

TCP2_deinterleaveExt(   aprioriMap1,
                        extrinsicsMap2,
                        interleaverTable,
                        numExt);

<...MAP 1 decode...>
...
```

14.2.10 TCP2_interleaveExt

void TCP2_interleaveExt	**(** **TCP2_ExtrinsicData***	***aprioriMap2*,**
	const TCP2_ExtrinsicData *	***extrinsicsMap1*,**
	const Uint16 *	***interleaverTable*,**
	Uint32	***numExt***
	)	

Description
This function interleaves the MAP1 extrinsics data to generate apriori data for the MAP2 decode. This function is for used in performing shared processing.

Arguments

aprioriMap2	Apriori data for MAP2 decode
extrinsicsMap1	Extrinsics data
interleaverTable	Interleaver data table
numExt	Number of Extrinsics

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The aprioriMap2 argument passed is modified to contain the interleaved data.

Example

```
extern TCP2_ExtrinsicData     *aprioriMap2;
extern TCP2_ExtrinsicData     *extrinsicsMap1;
extern Uint16                 *interleaverTable;
Uint32 numExt = 20800;

<...MAP 1 decode...>
TCP2_interleaveExt(     aprioriMap2,
                        extrinsicsMap1,
                        interleaverTable,
                        numExt);
<...MAP 2 decode...>
...
```

14.2.11 TCP2_depunctInputs

void TCP2_depunctInputs	**(**	**Uint32**	***frameLen*,**
		TCP2_UserData *	***inputData*,**
		TCP2_Rate	***rate*,**
		Uint32	***scalingFact*,**
		TCP2_InputData *	***depunctData***
	)		

Description
This function scales and sorts input data of any code rate into a code rate 1/5 format.

Arguments

frameLen	Input data length in bytes
inputData	Input data

```
rate                Input data code rate

scalingFact         Scaling factor

depunctData         Depunctured data
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The depunctData argument passed, to contain the data depunctured to rate 1/5

Example

```
      TCP2_UserData inputData []= { 0x2f,
                                    0x31,
                                    0x30,
                                    0x20,
                                    0x32,
                                    0x27,
                                    0x30,
                                    0x0d,
                                    0x10,
                                    0x3f,
                                    0x18,
                                    0x3b };
Uint32            rate = TCP2_RATE_1_4;
Uint32            frameLength = 40;
TCP2_InputData *  depunctData;
Uint32            scalingFact;

TCP2_depunctInputs (frameLength,
                  inputData,
                  rate
                  scalingFact,
                  depunctData);
```

14.2.12 TCP2_calculateHd

void TCP2_calculateHd	**(**	**const TCP2_ExtrinsicData***	***extrinsicsMap1*,**
		const TCP2_ExtrinsicData *	***apriori*,**
		const TCP2_UserData *	***channelData*,**
		Uint32 *	***hardDecisions*,**
		Uint16	***numExt***

)

Description
This function calculates the hard decisions following multiple MAP decodings in shared processing mode.

Arguments

`extrinsicsMap1`	Extrinsics data following MAP1 decode
`apriori`	Apriori data following MAP2 decode
`channelData`	Input channel data
`hardDecisions`	Hard decisions
`numExt`	Number of extrinsics

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The hardDecisions argument passed, to contain the calculated hard decisions

Example
```
extern TCP2_ExtrinsicData     *apriori;
extern TCP2_ExtrinsicData     *extrinsicsMap1;
Uint32                        numExt = 20800;
extern TCP2_UserData          *channelData;
extern Uint32                 *hardDecisions;

<...Iterate through MAP1 and MAP2 decodes...>
TCP2_calculateHd( extrinsicsMap1,
                  apriori,
                  channelData,
                  hardDecisions,
                  numExt);
...
```

14.2.13 TCP2_demuxInput

void TCP2_demuxInput	**(**	**Uint32**	***rate*,**
		Uint32	***frameLen*,**
		const TCP2_UserData *	***input*,**
		const Uint16 *	***interleaver*,**

TCP2_ExtrinsicData* ***nonInterleaved*,**
TCP2_ExtrinsicData* ***interleaved***
)

Description
This function splits the input data into two working sets. One set contains the non-interleaved input data and is used with the MAP 1 decoding. The other contains the interleaved input data and is used with the MAP2 decoding. This function is used in shared processing mode.

Arguments

`rate`	`TCP data code rate`
`frameLen`	`Frame length`
`input`	`Input channel data`
`interleaver`	`Interleaver data table`
`nonInterleaved`	`Non Interleaved data for SP MAP0`
`interleaved`	`Interleaved data for SP MAP1`

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
The nonInterleaved argument, to contain the non-interleaved data and the interleaved argument, to contain the interleaved data.

Example
```
extern TCP2_ExtrinsicData     *interleaved;
extern TCP2_ExtrinsicData     *nonInterleaved;
Uint32                        frameLen = 20800;
extern TCP2_UserData          *inputData;
extern Uint16                 *interleaver;

TCP2_demuxInput ( TCP2_RATE_1_4,
                  frameLen,
                  inputData,
                  interleaver,
                  interleaved,
                  nonInterleaved);
...
```

INLINE FUNCTIONS

14.2.14 TCP2_normalCeil

CSL_IDEF_INLINE Uint32 TCP2_normalCeil **(Uint32** ***val1*****,**
Uint32 ***val2***
)

Description
Returns the value rounded to the nearest integer, greater than or equal to (val1/val2).

Arguments

```
val1        Value to be augmented.

val2        Value by which val1 must be divisible.
```

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32              frameLen = 51200;
Uint32              numSubFrame;

// to calculate the number of sub frames for SP mode
numSubFrame = TCP2_normalCeil(frameLen,
                              TCP2_SUB_FRAME_SIZE_MAX);
...
```

14.2.15 TCP2_ceil

CSL_IDEF_INLINE Uint32 TCP2_ceil **(Uint32** ***val*****,**
Uint32 ***pwr2***
)

Description
Returns the value rounded to the nearest integer, greater than or equal to (val/(2^pwr2)).

Arguments

```
val         Value to be augmented.

pwr2        The power of two by which val must be divisible.
```

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32     val1 = 512;
Uint32     val2 = 4;
Uint32     val3;

val3 = TCP2_ceil(val1, val2);
...
```

14.2.16 TCP2_setExtScaling

CSL_IDEF_INLINE Uint32 TCP2_setExtScaling (**Uint8** ***extrVal1***, **Uint8** ***extrVal2***, **Uint8** ***extrVal3***, **Uint8** ***extrVal4***)

Description
This function formats individual bytes into a 32-bit word, which is used to set the extrinsic configuration registers.

Arguments

`extrVal1`	`Extrinsic scaling value 1`
`extrVal2`	`Extrinsic scaling value 2`
`extrVal3`	`Extrinsic scaling value 3`
`extrVal4`	`Extrinsic scaling value 4`

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
extern TCP2_Params       *configParams;
TCP2_ConfigIc            configIc;
configIc.ic12 = TCP2_setExtScaling(configParams->extrScaling [0],
                                   configParams->extrScaling [1],
                                   configParams->extrScaling [2],
                                   configParams->extrScaling [3]);
...
```

14.2.17 TCP2_makeTailArgs

CSL_IDEF_INLINE Uint32 TCP2_makeTailArgs **(Uint8** ***byte17_12*****,**
Uint8 ***byte11_6*****,**
Uint8 ***byte5_0***
)

Description
This function formats individual bytes into a 32-bit word, which is used to set the tail bits configuration registers.

Arguments

```
byte17_12   Byte to be placed in bits 17-12 of the 32-bit value

byte11_6    Byte to be placed in bits 11-6 of the 32-bit value

byte5_0     Byte to be placed in bits 5-0 of the 32-bit value
```

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
extern TCP2_UserData  *userData;
TCP2_ConfigIc         configIc;
TCP2_UserData         *xabData;
Uint32                frameLen = 40;

xabData = &userData [frameLen];

configIc.ic6 = TCP2_makeTailArgs( xabData[10],
```

```
                                          xabData[8],
                                          xabData[6]);
      ...
```

14.2.18 TCP2_getAccessErr

CSL_IDEF_INLINE Uint32 TCP2_getAccessErr (void)

Description
This function returns the ACC bit value of the TCPERR register indicating whether an invalid access has been made to the TCP during operation.

0 - No error
1 - TCP rams (syst, parities, hard decisions, extrinsics, aprioris) access is not allowed in state 1. This causes an error interrupt to occur.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_getAccessErr()) {
...
}
```

14.2.19 TCP2_getErr

CSL_IDEF_INLINE Uint32 TCP2_getErr (void)

Description
This function returns the ERR bit value of the TCPERR register indicating whether an error has occurred during TCP operation.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition

None

Modifies
None

Example

```
if (TCP2_getErr()) {
...
}
```

14.2.20 TCP2_getTcpErrors

CSL_IDEF_INLINE Uint32 TCP2_getTcpErrors (void)

Description
This function returns the TCPERR register value.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None
Modifies
None

Example

```
if (TCP2_getTcpErrors()) {
...
}
```

14.2.21 TCP2_getFrameLenErr

CSL_IDEF_INLINE Uint32 TCP2_getFrameLenErr (void)

Description
This function returns a boolean value indicating whether an invalid frame length has been programmed in the TCP during operation.

0 - no error.
1 - (SA mode) frame length < 40 or frame length > 20730.

- (SP mode) frame length < 256 or frame length > 20480 and f%256!=0 for the first or middle subframes.

- (SP mode) if f<128 or f>20480 for the last subframe.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getFrameLenErr()) {
...
}
```

14.2.22 TCP2_getProlLenErr

CSL_IDEF_INLINE Uint32 TCP2_getProlLenErr (void)

Description
This function returns the P bit value indicating whether an invalid prolog length has been programmed into the TCP.
0 - no error
1 - Prolog length < 4 or > 48

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getProlLenErr()) {
    ...
}
```

14.2.23 TCP2_getSubFrameErr

CSL_IDEF_INLINE Uint32 TCP2_getSubFrameErr (void)

Description
This function returns a boolean value indicating whether the sub-frame length programmed into the TCP is invalid.
0 - no error
1 - sub-frame length > 20480 (SP mode)

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getSubFrameErr()) {
    ...
}
```

14.2.24 TCP2_getRelLenErr

CSL_IDEF_INLINE Uint32 TCP2_getRelLenErr (void)

Description
This function returns the R bit value indicating whether an invalid reliability length has been programmed into the TCP. The reliability length must be 40 < RL < 128 for SA Mode, or must be 128 during first/middle subframes of SP mode, and must be > 64 in the last subframe.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getRelLenErr()) {
}
```

14.2.25 TCP2_getSnrErr

CSL_IDEF_INLINE Uint32 TCP2_getSnrErr (void)

Description
This function returns the SNR bit value indicating whether the SNR threshold exceeded 100 (1) or not (0).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getSnrErr()) {
...
}
```

14.2.26 TCP2_getInterleaveErr

CSL_IDEF_INLINE Uint32 TCP2_getInterleaveErr (void)

Description
This function returns the INTER value bit indicating whether the TCP was incorrectly programmed to receive an interleaver table. An interleaver table can only be sent when operating in standalone mode. This bit(1) indicates if an interleaver table was sent when in shared processing mode.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getInterleaveErr()) {
    ...
}
```

14.2.27 TCP2_getOutParmErr

CSL_IDEF_INLINE Uint32 TCP2_getOutParmErr (void)

Description
This function returns the OP bit value (1) indicating whether the TCP was programmed to transfer output parameters in shared processing mode. The output parameters are only valid when operating in standalone mode.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getOutParmErr()) {
    ...
}
```

14.2.28 TCP2_getMaxMinErr

CSL_IDEF_INLINE Uint32 TCP2_getMaxMinErr (void)

Description
This function returns the MAXMINITER bit value indicating whether the TCP was programmed with the minimum iterations value greater than the maximum iterations.
0 = no error
1 = min_iter > max_iter

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getMaxMinErr()) {
...
}
```

14.2.29 TCP2_getNumIt

CSL_IDEF_INLINE Uint32 TCP2_getNumIt (void)

Description
This function returns the number of decoded iterations of the TCP in standalone processing mode. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32  numIter;

numIter = TCP2_getNumIt();
...
```

14.2.30 TCP2_getSnrM1

CSL_IDEF_INLINE Uint32 TCP2_getSnrM1 (void)

Description
This function returns the 1st moment of SNR calculation. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 snrM1;
snrM1 = TCP2_getSnrM1();
...
```

14.2.31 TCP2_getSnrM2

CSL_IDEF_INLINE Uint32 TCP2_getSnrM2 (void)

Description
This function returns the second moment of SNR calculation. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32      snrM2;
snrM2 = TCP2_getSnrM2();
...
```

14.2.32 TCP2_getMap

CSL_IDEF_INLINE Uint32 TCP2_getMap (void)

Description
This function returns the active MAP of the TCP. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).
0 - MAP 0 is active 1 - MAP 1 is active

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 activeMap;

activeMap = TCP2_getMap();
...
```

14.2.33 TCP2_getMap0Err

CSL_IDEF_INLINE Uint32 TCP2_getMap0Err (void)

Description
This function returns the number of re-encode errors for MAP 0. This function reads the output parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 map0Err;
map0Err = TCP2_getMap0Err();
```

14.2.34 TCP2_getMap1Err

CSL_IDEF_INLINE Uint32 TCP2_getMap1Err (void)

Description
This function returns the number of re-encode errors for MAP 1. This function reads the output

parameters register. Alternatively, the EDMA can be used to transfer the output parameters following the hard decisions (recommended).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 map1Err;
map1Err = TCP2_getMap1Err();
...
```

14.2.35 TCP2_statRun

CSL_IDEF_INLINE Uint32 TCP2_statRun (void)

Description
This function returns a boolean status indicating whether the TCP MAP decoder is in state 0 or state 1-8 (running or not).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
while (!TCP2_statRun());
...
```

14.2.36 TCP2_statError

CSL_IDEF_INLINE Uint32 TCP2_statError (void)

Description
This function returns the ERR bit value of the TCPSTAT register indicating whether TCP has encountered an error in the input register configuration..

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statError()){
    ...
}
```

14.2.37 TCP2_statWaitIc

CSL_IDEF_INLINE Uint32 TCP2_statWaitIc (void)

Description
This function returns the WIC bit status indicating whether the TCP is waiting to receive new IC values.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statWaitIc()) {
    ...
}
```

14.2.38 TCP2_statWaitInter

CSL_IDEF_INLINE Uint32 TCP2_statWaitInter (void)

Description
This function returns the WINT status indicating whether the TCP is waiting to receive interleaver table data.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statWaitInter()) {
    ...
}
```

14.2.39 TCP2_statWaitSysPar

CSL_IDEF_INLINE Uint32 TCP2_statWaitSysPar (void)

Description
This function returns the WSP bit status indicating whether the TCP is waiting to receive systematic and parity data.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statWaitSysPar()){
    ...
}
```

14.2.40 TCP2_statWaitApriori

CSL_IDEF_INLINE Uint32 TCP2_statWaitApriori (void)

Description
This function returns the WAP bit status indicating whether the TCP is waiting to receive apriori data.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statWaitApriori()) {
    ...
}
```

14.2.41 TCP2_statWaitExt

CSL_IDEF_INLINE Uint32 TCP2_statWaitExt (void)

Description
This function returns the REXT bit status indicating whether the TCP is waiting for extrinsic data to be read.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_statWaitExt()) {
    ...
```

```
}
```

14.2.42 TCP2_statWaitHardDec

CSL_IDEF_INLINE Uint32 TCP2_statWaitHardDec (void)

Description
This function returns the RHD bit status indicating whether the TCP is waiting for the hard decisions data to be read.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_statWaitHardDec()) {
    ...
}
```

14.2.43 TCP2_statWaitOutParm

CSL_IDEF_INLINE Uint32 TCP2_statWaitOutParm (void)

Description
This function returns the ROP bit status indicating whether the TCP is waiting for the output parameters to be read.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_statWaitOutParm()) {
    ...
}
```

14.2.44 TCP2_statEmuHalt

CSL_IDEF_INLINE Uint32 TCP2_statEmuHalt (void)

Description
This function returns the emuhalt bit status indicating whether the TCP is halted due to emulation.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_statEmuHalt()) {
    ...
}
```

14.2.45 TCP2_statActMap

CSL_IDEF_INLINE Uint32 TCP2_statActMap (void)

Description
This function returns the active_map bit status of the TCPSTAT register indicating whether the TCP MAP 0 is active (0) or the TCP MAP 1 is active (1).

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 activeMap;
...
activeMap = TCP2_statActMap();
...
```

14.2.46 TCP2_statActState

CSL_IDEF_INLINE Uint32 TCP2_statActState (void)

Description
This function returns the active_state bit status indicating the active TCP MAP decoder state.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 activeState;
...
activeState = TCP2_statActState();
...
```

14.2.47 TCP2_statActIter

CSL_IDEF_INLINE Uint32 TCP2_statActIter (void)

Description
This function returns the active_iter bit status indicating the active TCP iteration.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition

None

Modifies
None

Example

```
Uint32 activeIter;
...
activeIter = TCP2_statActIter();
...
```

14.2.48 TCP2_statSnr

CSL_IDEF_INLINE Uint32 TCP2_statSnr (void)

Description
This function returns the snr_exceed bits, indicating whether the TCP MAP 0 or MAP 1 passed the SNR criteria in a particular iteration.
0 - All fail, 1 - MAP 1 failed, 2 - MAP 0 failed, 3 - All passed

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 snrExceed;
...
snrExceed = TCP2_statSnr();
...
```

14.2.49 TCP2_statCrc

CSL_IDEF_INLINE Uint32 TCP2_statCrc (void)

Description
This function returns the crc_pass bit boolean status indicating whether the TCP passed CRC check.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_statCrc()) {
    ...
}
```

14.2.50 TCP2_statTcpState

CSL_IDEF_INLINE Uint32 TCP2_statTcpState (void)

Description
This function returns the state of the TCP state machine for the standalone mode or shared processing mode.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 tcpState;
...
tcpState = TCP2_statTcpState();
...
```

14.2.51 TCP2_getExecStatus

CSL_IDEF_INLINE Uint32 TCP2_getExecStatus (void)

Description
This function returns the TCPSTAT register value.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 tcpStatus;

tcpStatus = TCP2_getExecStatus();
...
```

14.2.52 TCP2_getExtEndian

CSL_IDEF_INLINE Uint32 TCP2_getExtEndian (void)

Description
This function returns the value programmed into the TCP_END register for the extrinsics data indicating whether the data is in its native 8-bit format ('1') or consists of values packed in little endian format into 32-bit words ('0'). This should always be '0' for little endian operation.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getExtEndian()) {
}
...
```

14.2.53 TCP2_getInterEndian

CSL_IDEF_INLINE Uint32 TCP2_getInterEndian (void)

Description
Returns the value programmed into the TCP_END register for the interleaver table data indicating

whether the data is in its native 8-bit format ('1') or consists of values packed in little endian format into 32-bit words ('0'). This should always be '0' for little endian operation.

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getInterEndian()) {
   ...
}
```

14.2.54 TCP2_getSlpzvss

CSL_IDEF_INLINE Uint32 TCP2_getSlpzvss (void)

Description
This function gets the configuration of the internal control of the slpzvss.
0 = sleep mode disabled
1 = internal control of slpzvss enabled
Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (TCP2_getSlpzvss()) {
    ...
}
```

14.2.55 TCP2_getSlpzvdd

CSL_IDEF_INLINE Uint32 TCP2_getSlpzvdd (void)

Description
This function gets the configuration of the internal control of the slpzvdd.
0 = sleep mode disabled
1 = internal control of slpzvdd enabled

Arguments
None

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
if (TCP2_getSlpzvdd()) {
    ...
}
```

14.2.56 TCP2_setExtEndian

CSL_IDEF_INLINE void TCP2_setExtEndian (Uint32 *endianMode*)

Description
This function programs TCP to view the format of the extrinsics data as either native 8-bit format ('1') or values packed into 32-bit words in little endian format ('0'). This should always be '0' for little endian operation.

Arguments

`endianMode`	`Endian setting for extrinsics data`

Return Value
None

Pre Condition
None

Post Condition
TCPEND register bit for the extrinsics data is configured in the mode passed.

Modifies
TCPEND register

Example
```
TCP2_setExtEndian(1);
```

```
...
```

14.2.57 TCP2_setInterEndian

CSL_IDEF_INLINE void TCP2_setInterEndian (Uint32 *endianMode*)

Description
This function programs TCP to view the format of the interleaver data as either native 8-bit format ('1') or values packed into 32-bit words in little endian format ('0'). This should always be '0' for little endian operation.

Arguments

```
endianMode       Endian setting for interleaver data
```

Return Value
None

Pre Condition
None

Post Condition
TCPEND register bit for the interleaver data is configured in the mode passed.

Modifies
TCPEND register

Example

```
TCP2_setInterEndian(1);
...
```

14.2.58 TCP2_setNativeEndian

CSL_IDEF_INLINE void TCP2_setNativeEndian (void)

Description
This function programs the TCP to view the format of all data as native 8/16-bit format. This should only be used when running in big endian mode.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
TCPEND register configured to native mode for all data.

Modifies
TCPEND register

Example

```
TCP2_setNativeEndian();
...
```

14.2.59 TCP2_setPacked32Endian

CSL_IDEF_INLINE void TCP2_setPacked32Endian (void)

Description
This function programs the TCP to view the format of all data as packed data in 32-bit words. This should always be used when running in little endian mode and should be used in big endian mode only if the CPU is formatting the data.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
TCPEND register configured to packed 32 mode for all data.

Modifies
TCPEND register

Example

```
TCP2_setPacked32Endian();
...
```

14.2.60 TCP2_start

CSL_IDEF_INLINE void TCP2_start (void)

Description
This function starts the TCP by writing a ‘1h’ to the EXEINST field of the TCPEXE register. See also TCP2_debug(), TCP2_debugStep() and TCP2_debugComplete().

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
TCP state machine starts executing.

Modifies
TCPEXE register

Example

```
TCP2_start();
...
```

14.2.61 TCP2_debug

CSL_IDEF_INLINE void TCP2_debug (void)

Description
This function puts the TCP into debug mode by writing '4h' to the EXEINST field of the TCPEXE register. Normal initialization is performed and TCP waits in MAP state 0 for Debug Step or Debug Complete to be performed. See also TCP2_start(), TCP2_debugStep() and TCP2_debugComplete().

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
EXEINST field of the TCPEXE register is configured.

Modifies
TCPEXE register

Example

```
TCP2_debug();
...
```

14.2.62 TCP2_debugStep

CSL_IDEF_INLINE void TCP2_debugStep (void)

Description
This function executes one MAP decode and waits in state 6 when the TCP is in Debug mode, by writing '5h' to the EXEINST field of the TCPEXE register. See also TCP2_start(),TCP2_debug(), and TCP2_debugComplete().

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
EXEINST field of the TCPEXE register is configured

Modifies
TCPEXE register

Example
```
TCP2_debugStep();
...
```

14.2.63 TCP2_debugComplete

CSL_IDEF_INLINE void TCP2_debugComplete (void)

Description
This function executes the remaining MAP decodes when the TCP is in Debug mode, by writing '6h' to the EXEINST field of the TCPEXE register. See also TCP2_start(), TCP2_debug(), and TCP2_debugStep().

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
EXEINST field of the TCPEXE register is configured.

Modifies
TCPEXE register

Example
```
TCP2_debugComplete();
...
```

14.2.64 TCP2_reset

CSL_IDEF_INLINE void TCP2_reset (void)

Description
This function performs a soft reset of all TCP registers except for TCPEXE and TCPEND registers.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
Performs soft reset of TCP2.

Modifies
TCPEXE register

Example
```
TCP2_reset();
...
```

14.2.65 TCP2_setSlpzvdd

CSL_IDEF_INLINE void TCP2_setSlpzvdd (Uint32 *slpzvddCtrl*)

Description
This function enables/disables the internal control of the slpzvdd.

Arguments

```
slpzvddCtrl      Enable/disable configuration of the slpzvdd
```

Return Value
None

Pre Condition
None

Post Condition
TCPEND register configured with the value passed.

Modifies
TCPEND register

Example
```
TCP2_setSlpzvdd(1);
...
```

14.2.66 TCP2_setSlpzvss

CSL_IDEF_INLINE void TCP2_setSlpzvss (Uint32 *slpzvssCtrl*)

Description
This function enables/disables the internal control of the slpzvss.

Arguments

```
slpzvssCtrl      Enable/disable configuration of the slpzvss
```

Return Value
None

Pre Condition
None

Post Condition
TCPEND register configured with the value passed.

Modifies
TCPEND register

Example
```
TCP2_setSlpzvss(1);
...
```

14.2.67 TCP2_getIcConfig

CSL_IDEF_INLINE void TCP2_getIcConfig (TCP2_ConfigIc * *config*)

Description
This function reads the input configuration values currently programmed into the TCP.

Arguments

```
config        TCP configuration structure to hold the read values
```

Return Value
None

Pre Condition
None

Post Condition
config structure contains the TCP input configuration values.

Modifies
None

Example
```
TCP2_ConfigIc configIc;
TCP2_getIcConfig(&configIc);
...
```

14.2.68 TCP2_icConfig

CSL_IDEF_INLINE void TCP2_icConfig (TCP2_ConfigIc * *config*)

Description
Programs the TCP with the input configuration values passed in the TCP2_ConfigIc structure. This is not the recommended means by which to program the TCP, as it is more efficient to transfer the IC values using the EDMA.

Arguments

```
config        TCP configuration structure containing the values
              to be programmed
```

Return Value
None

Pre Condition

None

Post Condition
TCP input configuration registers are programmed with the values passed.

Modifies
TCP input configuration registers.

Example

```
TCP2_ConfigIc configIc;
configIc.ic0 = 0x00283300;
configIc.ic1 = 0x00270000;
configIc.ic2 = 0x00080118;
configIc.ic3 = 0x00000011;
configIc.ic4 = 0x00000100;
configIc.ic5 = 0x00000000;
configIc.ic6 = 0x00032c2f;
configIc.ic7 = 0x00027831;
configIc.ic8 = 0x00000000;
configIc.ic9 = 0x00018430;
configIc.ic10 = 0x0003bfcd;
configIc.ic11 = 0x00000000;
configIc.ic12 = 0x00820820;
configIc.ic13 = 0x00820820;
configIc.ic14 = 0x00820820;
configIc.ic15 = 0x00820820;

TCP2_icConfig(&configIc);
...
```

14.2.69 TCP2_icConfigArgs

CSL_IDEF_INLINE void TCP2_icConfigArgs **(** **Uint32** ***ic0*****,**
Uint32 ***ic1*****,**
Uint32 ***ic2*****,**
Uint32 ***ic3*****,**
Uint32 ***ic4*****,**
Uint32 ***ic5*****,**
Uint32 ***ic6*****,**
Uint32 ***ic7*****,**
Uint32 ***ic8*****,**
Uint32 ***ic9*****,**
Uint32 ***ic10*****,**
Uint32 ***ic11*****,**

Uint32 ***ic12*,**

Uint32 ***ic13*,**

Uint32 ***ic14*,**

Uint32 ***ic15***

)

Description
Programs the TCP with the input configuration values passed. This is not the recommended means by which to program the TCP, as it is more efficient to transfer the IC values using the EDMA.

Arguments

```
ic0        TCP input configuration register 0 value
ic1        TCP input configuration register 1 value
ic2        TCP input configuration register 2 value
ic3        TCP input configuration register 3 value
ic4        TCP input configuration register 4 value
ic5        TCP input configuration register 5 value
ic6        TCP input configuration register 6 value
ic7        TCP input configuration register 7 value
ic8        TCP input configuration register 8 value
ic9        TCP input configuration register 9 value
ic10       TCP input configuration register 10 value
ic11       TCP input configuration register 11 value
ic12       TCP input configuration register 12 value
ic13       TCP input configuration register 13 value
ic14       TCP input configuration register 14 value
ic15       TCP input configuration register 15 value
```

Return Value
None

Pre Condition
None

Post Condition

TCP input configuration registers are programmed with the values passed.

Modifies
TCP input configuration registers

Example

```
Uint32 ic0, ic1, ic2, ic3, ic4, ic5, ic6, ic7, ic8, ic9, ic10,
       ic11, ic12, ic13, ic14, ic15;
ic0 = 0x00283300;
ic1 = 0x00270000;
ic2 = 0x00080118;
ic3 = 0x00000011;
ic4 = 0x00000100;
ic5 = 0x00000000;
ic6 = 0x00032c2f;
ic7 = 0x00027831;
ic8 = 0x00000000;
ic9 = 0x00018430;
ic10 = 0x0003bfcd;
ic11 = 0x00000000;
ic12 = 0x00820820;
ic13 = 0x00820820;
ic14 = 0x00820820;
ic15 = 0x00820820;
TCP2_icConfigArgs(ic0, ic1, ic2, ic3, ic4, ic5, ic6, ic7,
                  ic8, ic9, ic10, ic11, ic12, ic13, ic14, ic15);
...
```

14.3 Data Structures

This section lists the data structures available in the TCP2 module.

14.3.1 TCP2_ConfigIc

Detailed Description

The TCP input configuration structure holds all the configuration values that are to be transferred to the TCP via the EDMA

Field Documentation

Uint32 TCP2_ConfigIc::ic0
TCP input configuration word 0 value

Uint32 TCP2_ConfigIc::ic1
TCP input configuration word 1 value

Uint32 TCP2_ConfigIc::ic2
TCP input configuration word 2 value

Uint32 TCP2_ConfigIc::ic3
TCP input configuration word 3 value

Uint32 TCP2_ConfigIc::ic4
TCP input configuration word 4 value

Uint32 TCP2_ConfigIc::ic5
TCP input configuration word 5 value

Uint32 TCP2_ConfigIc::ic6
TCP input configuration word 6 value

Uint32 TCP2_ConfigIc::ic7
TCP input configuration word 7 value

Uint32 TCP2_ConfigIc::ic8
TCP input configuration word 8 value

Uint32 TCP2_ConfigIc::ic9
TCP input configuration word 9 value

Uint32 TCP2_ConfigIc::ic10
TCP input configuration word 10 value

Uint32 TCP2_ConfigIc::ic11
TCP input configuration word 11 value

Uint32 TCP2_ConfigIc::ic12
TCP input configuration word 12 value

Uint32 TCP2_ConfigIc::ic13
TCP input configuration word 13 value

Uint32 TCP2_ConfigIc::ic14
TCP input configuration word 14 value

Uint32 TCP2_ConfigIc::ic15
TCP input configuration word 15 value

14.3.2 TCP2_Params

Detailed Description
The TCP parameters structure holds all the information concerning the user channel. These values are used to generate the appropriate input configuration values for the TCP.

Field Documentation

Uint8 TCP2_Params::crcLen
CRC polynomial length

Uint32 TCP2_Params::crcPoly
CRC polynomial

Uint8 TCP2_Params::extrScaling[16]
Extrinsic scaling factors

Uint32 TCP2_Params::frameLen
Frame length

TCP2_InputSign **TCP2_Params::inputSign**
The sign of the input data (+/-)

Uint32 TCP2_Params::intFlag
Interleaver write flag

TCP2_Map **TCP2_Params::map**
TCP2 shared processing MAP

Uint32 TCP2_Params::maxIter
Maximum number of iterations

Bool TCP2_Params::maxStarEn
Enable/disable the max star

Uint8 TCP2_Params::minIter
Minimum number of iterations to be executed

TCP2_Mode **TCP2_Params::mode**
TCP mode

Uint8 TCP2_Params::numCrcPass
Number of passed CRC iterations required before decoder termination

TCP2_NumSW **TCP2_Params::numSlideWin**
Number of sliding windows per sub block

Uint32 TCP2_Params::numSubBlock
Number of sub blocks

Uint32 TCP2_Params::outParmFlag
Output parameters read flag

TCP2_OutputOrder TCP2_Params::outputOrder
The bit ordering of the output data

Bool TCP2_Params::prologRedEn
Enable/disable the prolog reduction

Uint32 TCP2_Params::prologSize
Prolog length

TCP2_Rate TCP2_Params::rate
TCP code rate

Uint32 TCP2_Params::relLen
Reliability length

Uint32 TCP2_Params::snr
SNR threshold used for stopping test

TCP2_Standard TCP2_Params::standard
TCP standard

14.3.3 TCP2_BaseParams

Detailed Description
The TCP base parameters structure is used to set up the TCP programmable parameters. The user as to create the object and pass it to the TCP2_genParams() function which returns the TCP2_Params structure.

Field Documentation

Uint8 TCP2_BaseParams::crcLen
CRC polynomial length

Uint32 TCP2_BaseParams::crcPoly
CRC polynomial

Uint8 TCP2_BaseParams::extrScaling[16]
Extrinsic scaling factors

Uint32 TCP2_BaseParams::frameLen
Frame length

TCP2_InputSign TCP2_BaseParams::inputSign
The sign of the input data (+/-)

Uint32 TCP2_BaseParams::intFlag
Interleaver write flag

TCP2_Map TCP2_BaseParams::map
TCP shared processing MAP

Uint32 TCP2_BaseParams::maxIter
Maximum number of iterations

Bool TCP2_BaseParams::maxStarEn
Enable/disable the max star

Uint8 TCP2_BaseParams::minIter
Minimum number of iterations to be executed

Uint8 TCP2_BaseParams::numCrcPass
Number of passed CRC iterations required before decoder termination

Uint32 TCP2_BaseParams::outParmFlag
Output parameters read flag

TCP2_OutputOrder TCP2_BaseParams::outputOrder
The bit ordering of the output data

Bool TCP2_BaseParams::prologRedEn
Enable/disable the prolog reduction

Uint32 TCP2_BaseParams::prologSize
Prolog length

TCP2_Rate TCP2_BaseParams::rate
TCP code rate

Uint32 TCP2_BaseParams::snr
SNR threshold used for stopping test

TCP2_Standard TCP2_BaseParams::standard
TCP decoder standards

14.4 Enumerations

This section lists the enumerations available in the TCP2 module.

14.4.1 TCP2_InputSign

enum TCP2_InputSign
Enum for the input sign values

Enumeration values:

TCP2_INPUT_SIGN_POSITIVE	Multiply the channel input by +1
TCP2_INPUT_SIGN_NEGATIVE	Multiply the channel input by -1

14.4.2 TCP2_OutputOrder

enum TCP2_OutputOrder
Enum for the output order values

Enumeration values:

TCP2_OUT_ORDER_0_31	Order of the bits in the output data is 0-31
TCP2_OUT_ORDER_31_0	Order of the bits in the output data is 31-0

14.5 Macros

#define tcp2CfgRegs ((CSL_Tcp2CfgRegs*)CSL_TCP2_CFG_REGS)
Address of the TCP2 configuration registers

#define tcp2Regs ((CSL_Tcp2Regs*)CSL_TCP2_0_REGS)
Address of the TCP2 registers

#define TCP2_FIRST_SF CSL_TCP2_TCPIC0_OPMOD_SP_FF
TCP shared processing, first sub frame

#define TCP2_FLEN_MAX 20730
TCP maximum standalone mode frame size

#define TCP2_LAST_SF CSL_TCP2_TCPIC0_OPMOD_SP_LF
TCP shared processing, last sub frame

#define TCP2_MAP_MAP1 0
TCP shared processing non interleaved MAP

#define TCP2_MAP_MAP2 1
TCP shared processing interleaved MAP

#define TCP2_MIDDLE_SF CSL_TCP2_TCPIC0_OPMOD_SP_MF
TCP shared processing, middle sub frame

#define TCP2_MODE_SA CSL_TCP2_TCPIC0_OPMOD_SA
TCP stand alone mode

#define TCP2_MODE_SP 1
TCP shared processing mode

#define TCP2_RATE_1_2 CSL_TCP2_TCPIC0_RATE_1_2
Define for TCP code rate 1/2

#define TCP2_RATE_1_3 CSL_TCP2_TCPIC0_RATE_1_3
Define for TCP code rate 1/3

#define TCP2_RATE_1_4 CSL_TCP2_TCPIC0_RATE_1_4
Define for TCP code rate 1/4

#define TCP2_RATE_1_5 CSL_TCP2_TCPIC0_RATE_1_5
Define for TCP code rate 1/5

#define TCP2_RATE_3_4 CSL_TCP2_TCPIC0_RATE_3_4
Define for TCP code rate 3/4

#define TCP2_RLEN_MAX 128
TCP maximum reliability length

#define TCP2_STANDARD_3GPP 0
Decoder standard 3GPP

#define TCP2_STANDARD_IS2000 1
Decoder standard IS2000

#define TCP2_SUB_FRAME_SIZE_MAX 20480
TCP maximum sub frame size

#define TCP2_SW_G128 CSL_TCP2_TCPIC0_NUMSW_G_128
Number of sliding windows per block is > 128

#define TCP2_SW_LEQ128 CSL_TCP2_TCPIC0_NUMSW_LEQ_128
Number of sliding windows per block is <= 128

14.6 Typedefs

typedef Uint8 TCP2_ExtrinsicData
This data type is used to represent the TCP extrinsic data

typedef Uint32 TCP2_InputData
This data type is used to represent the TCP input data

typedef Uint8 TCP2_Map
This data type is used to define the TCP map (1,2)

typedef Uint8 TCP2_Mode
This data type is used to define the TCP mode (stand alone or shared processing)

typedef Uint8 TCP2_NumSW
This data type is used to define the number of sliding windows per block

typedef Uint8 TCP2_Rate
This data type is used to define the TCP code rates (1/2, 1/3, 1/4, 1/5, 3/4)

typedef Uint8 TCP2_Standard
This data type is used to define the TCP standards

typedef Int8 TCP2_TailData
This data type is used to represent the TCP tail data

typedef Uint8 TCP2_UserData
This data type is used to represent the TCP data

Chapter 15
TIMER Module

Topics

15.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within TIMER module. The timer can be configured as a general-purpose 64-bit timer, dual general-purpose 32-bit timers, or a watchdog timer. When configured as a dual 32-bit timers, each half can operate in conjunction (chain mode) or independently (unchained mode) of each other. The timer can be configured in one of three modes using the timer mode (TIMMODE) bits in the timer global control register (TGCR): a 64-bit general-purpose (GP) timer, dual 32-bit timers (TIMLO and TIMHI), or a watchdog timer. When configured as dual 32-bit timers, each half can operate dependently (chain mode) or independently (unchained mode) of each other. At reset, the timer is configured as a 64-bit GP timer. The watchdog timer function can be enabled if desired, via the TIMMODE bits in timer global control register (TGCR) and WDEN bit in the watchdog timer control register WDTCR). Once the timer is configured as a watchdog timer, it cannot be re-configured as a regular timer until a device reset occurs. The timer has one input pin (TINPL) and one output pin (TOUTL). The timer control register (TCR) controls the function of the input and output pin.

The timers can be used to: time events, count events, generate pulses, interrupt the CPU, and send synchronization events to the EDMA.

15.2 Functions

This section lists the functions available in the TIMER module.

15.2.1 CSL_tmrInit

CSL_Status CSL_tmrInit (CSL_TmrContext * *pContext*)

Description
This is the initialization function for the TIMER CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context. As timer doesn't have any
            context based information user is expected to pass
            NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK - Always returns`

Pre Condition
None

Post Condition
The CSL for timer is initialized.

Modifies
None

Example

```
CSL_Status          status;
...
status = CSL_tmrInit(NULL);
...
```

15.2.2 CSL_tmrOpen

CSL_TmrHandle CSL_tmrOpen (CSL_TmrObj* *pTmrObj*, CSL_InstNum *tmrNum*, CSL_TmrParam * *pTmrParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the TIMER instance and returns a handle to the instance. The open call sets up the data structures for the particular instance of TIMER device. The device can be re-opened anytime after it has been normally closed if so required. The handle returned by this call is input argument for rest of the TIMER CSL APIs.

Arguments

```
pTmrObj          Pointer to timer object.

tmrNum           Instance of timer CSL to be opened.
                 There are two instance of the timer
                 available. So, the value for this parameter will
                 be based on the instance.

pTmrParam        Module specific parameters

pStatus          Status of the function call
```

Return Value
`CSL_TmrHandle`

- Valid timer handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The TIMER must be successfully initialized via CSL_tmr*Init*() before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- CSL_SOK - Valid timer handle is returned
- CSL_ESYS_FAIL - The timer instance is invalid
- CSL_ESYS_INVPARAMS - The object structure is not properly initialized

2. Timer object structure is populated.

Modifies
1. Timer object structure
2. The status variable

Example
```
CSL_Status              status;
CSL_TmrObj              tmrObj;
CSL_TmrHandle           hTmr;
...
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);
...
```

15.2.3 CSL_tmrClose

CSL_Status CSL_tmrClose (CSL_TmrHandle *hTmr*)

Description
This function closes the specified instance of TIMER. CSL for the timer instance need to be reopened before using any timer CSL API.

Arguments

```
hTmr          Pointer to the object that holds reference to the
```

```
instance of TIMER
```

Return Value
`CSL_Status`

- `CSL_SOK` - Timer close successful
- `CSL_ESYS_BADHANDLE` - The handle passed is invalid

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling *CSL_tmrClose()*.

Post Condition
The timer CSL APIs can not be called until the timer CSL is reopened again using CSL_tmrOpen()

Modifies
Obj structure values for the instance

Example
```
CSL_TmrHandle   hTmr;
CSL_Status      status;
...
status = CSL_tmrClose(hTmr);
...
```

15.2.4 CSL_tmrHwSetup

CSL_Status CSL_tmrHwSetup **(** [CSL_TmrHandle]() ***hTmr*****,**
[CSL_TmrHwSetup]() ***** ***hwSetup***
)

Description
It configures the timer instance registers as per the values passed in the hardware setup structure.

Arguments
```
hTmr            Handle to the TIMER instance

hwSetup         Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Hardware structure is not properly initialized

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition
The specified instance will be setup according the hardware setup parameters.

Modifies
Timer registers for the specified instance

Example

```
CSL_Status          status;
CSL_TmrHwSetup      hwSetup;
CSL_TmrHandle       hTmr;

hwSetup.tmrTimerPeriodLo = 0x100;
hwSetup.tmrTimerPeriodLo = 0x100;

...
status = CSL_tmrHwSetup(hTmr, &hwSetup);
...
```

15.2.5 CSL_tmrHwControl

CSL_Status CSL_tmrHwControl **(** [CSL_TmrHandle] ***hTmr*****,**
[CSL_TmrHwControlCmd] ***cmd*****,**
void * ***arg***
)

Description
This function performs various control operations on the timer instance, based on the command passed.

Arguments

`hTmr`	`Handle to the timer instance`
`cmd`	`Operation to be performed on the timer`
`arg`	`Optional argument as per the control command`

Return Value
`CSL_Status`

- `CSL_SOK` - Command execution successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition

Registers of the timer instance are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
TIMER Registers

Example

```
CSL_Status     status;
CSL_TmrHandle  hTmr;

...
status = CSL_tmrHwControl(hTmr, CSL_TMR_CMD_START_TIMLO, NULL);
...
```

15.2.6 CSL_tmrGetHwStatus

CSL_Status CSL_tmrGetHwStatus	**(**	**CSL_TmrHandle**	***hTmr*,**
		CSL_TmrHwStatusQuery	***query*,**
		void *	***response***
	)		

Description
This function is used to get the value of various parameters of the timer instance. The value returned depends on the query passed.

Arguments

`hTmr`	`Handle to the timer instance`
`query`	`Query to be performed`
`response`	`Pointer to buffer to return the data requested by the query passed`

Return Value
`CSL_Status`

- `CSL_SOK` - Successful completion of the query
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - Query command not supported
- `CSL_ESYS_INVPARAMS` – Invalid parameter

Pre Condition

Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_Status    status;
CSL_TmrHandle hTmr;
Uint32        count;
...
status = CSL_tmrGetHwStatus(      hTmr,
                                  CSL_TMR_QUERY_COUNT_LO,
                                  &count);
...
```

15.2.7 CSL_tmrHwSetupRaw

CSL_Status CSL_tmrHwSetupRaw	**(**	CSL_TmrHandle	***hTmr*,**
		CSL_TmrConfig *****	***config***
	)		

Description
This function initializes the device registers with the register-values provided through the config data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values.

Arguments

`hTmr`	`Handle to the timer instance`
`config`	`Pointer to the config structure containing the device register values`

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration structure pointer is not properly initialized

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition
The registers of the specified timer instance will be setup according to the values passed through the Config structure.

Modifies
Hardware registers of the specified timer instance

Example

```
CSL_TmrHandle           hTmr;
```

```
CSL_TmrConfig            config = CSL_TMR_CONFIG_DEFAULTS;
CSL_Status               status;
...
status = CSL_tmrHwSetupRaw(hTmr, &config);
...
```

15.2.8 CSL_tmrGetHwSetup

CSL_Status CSL_tmrGetHwSetup **(** **CSL_TmrHandle** ***hTmr*****,** **CSL_TmrHwSetup *** ***hwSetup*** **)**

Description
This function gets the current setup of the TIMER. The status is returned through *CSL_tmrHwSetup*. The obtaining of status is the reverse operation of *CSL_tmrHwSetup()* function.

Arguments

```
hTmr        Handle to the timer instance

hwSetup     Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup retrieved
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Hardware structure is not properly initialized

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition
None

Modifies
Second parameter *hwSetup* value

Example

```
CSL_TmrHandle      hTmr;
CSL_Status         status;
CSL_TmrHwSetup     hwSetup;

...
status = CSL_tmrGetHwSetup(hTmr, &hwSetup);
...
```

15.2.9 CSL_tmrGetBaseAddress

CSL_Status CSL_tmrGetBaseAddress **(** **CSL_InstNum** ***tmrNum*****,**

[CSL_TmrParam](#) * *pTmrParam*,

[CSL_TmrBaseAddress](#) * *pBaseAddress*

)

Description

Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_tmrOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

```
tmrNum        Specifies the instance of the timer to be opened

pTmrParam     Timer module specific parameters

pBaseAddress  Pointer to base address structure containing base
              address details
```

Return Value

`CSL_Status`

- `CSL_SOK` - Successful on getting the base address of TIMER
- `CSL_ESYS_FAIL` - Timer instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid Parameters

Pre Condition

None

Post Condition

Base address structure is populated.

Modifies

1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status              status;
CSL_TmrBaseAddress      baseAddress;
...
status = CSL_tmrGetBaseAddress(CSL_TMR_1, NULL, &baseAddress);
...
```

15.3 Data Structures

This section lists the data structures available in the TIMER module.

15.3.1 CSL_TmrObj

Detailed Description
Timer object structure.

Field Documentation

CSL_InstNum CSL_TmrObj::perNum
Instance of Timer being referred by this object

CSL_TmrRegsOvly CSL_TmrObj::regs
Pointer to the register overlay structure of the Timer

15.3.2 CSL_TmrConfig

Detailed Description
Config-structure Used to configure the Timer using CSL_tmrHwSetupRaw(). This is a structure of register values, rather than a structure of register field values like CSL_TmrHwSetup.

Field Documentation

Uint32 CSL_TmrConfig::PRDHI
Timer Period Register High

Uint32 CSL_TmrConfig::PRDLO
Timer Period Register Low

Uint32 CSL_TmrConfig::TCR
Timer Control Register

Uint32 CSL_TmrConfig::TGCR
Timer Global Control Register

Uint32 CSL_TmrConfig::TIMHI
Timer Counter Register High

Uint32 CSL_TmrConfig::TIMLO
Timer Counter Register Low

Uint32 CSL_TmrConfig::WDTCR
Watchdog Timer Control Register

15.3.3 CSL_TmrContext

Detailed Description
Module specific context information. Present implementation of Timer CSL doesn't have any context information.

Field Documentation

Uint16 CSL_TmrContext::contextInfo

Context information of Timer CSL. The declaration is just a placeholder for future implementation.

15.3.4 CSL_TmrParam

Detailed Description
Module specific parameters. Present implementation of Timer CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_TmrParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

15.3.5 CSL_TmrHwSetup

Detailed Description
Hardware setup structure.

Field Documentation

CSL_TmrClksrc CSL_TmrHwSetup::tmrClksrcLo
CLKSRC determines the selected clock source for the timer

CSL_TmrClockPulse CSL_TmrHwSetup::tmrClockPulseHi
Clock/Pulse mode for timerHigh output

CSL_TmrClockPulse CSL_TmrHwSetup::tmrClockPulseLo
Clock/Pulse mode for timerLow output

CSL_TmrInvInp CSL_TmrHwSetup::tmrInvInpLo
Timer input inverter control. Only affects operation if CLKSRC=1, Timer Input pin

CSL_TmrInvOutp CSL_TmrHwSetup::tmrInvOutpHi
Timer output inverter control

CSL_TmrInvOutp CSL_TmrHwSetup::tmrInvOutpLo
Timer output inverter control

CSL_TmrIpGate CSL_TmrHwSetup::tmrIpGateLo
TIEN determines if the timer clock is gated by the timer input. Applicable only when CLKSRC=0

Uint8 CSL_TmrHwSetup::tmrPreScalarCounterHi
TIMHI pre-scalar counter specifies the count for TIMHI

CSL_TmrPulseWidth CSL_TmrHwSetup::tmrPulseWidthHi
Pulse width. Used in pulse mode (C/P_=0) by the timer

CSL_TmrPulseWidth CSL_TmrHwSetup::tmrPulseWidthLo
Pulse width. Used in pulse mode (C/P_=0) by the timer

Uint32 CSL_TmrHwSetup::tmrTimerCounterHi
32-bit load value to be loaded to Timer Counter Register High

Uint32 CSL_TmrHwSetup::tmrTimerCounterLo
32-bit load value to be loaded to Timer Counter Register Low

CSL_TmrMode CSL_TmrHwSetup::tmrTimerMode
Configures the Timer in GP mode or in general purpose timer mode or Dual 32 bit timer mode

Uint32 CSL_TmrHwSetup::tmrTimerPeriodHi
32-bit load value to be loaded to Timer Period Register High

Uint32 CSL_TmrHwSetup::tmrTimerPeriodLo
32-bit load value to be loaded to Timer Period Register low

15.3.6 CSL_TmrBaseAddress

Detailed Description
This structure contains the base-address information for the peripheral instance.

Field Documentation

CSL_TmrRegsOvly CSL_TmrBaseAddress::regs
Base-address of the configuration registers of the peripheral

15.4 Enumerations

This section lists the enumerations available in the TIMER module.

15.4.1 CSL_TmrHwControlCmd

enum CSL_TmrHwControlCmd

This enum describes the commands used to control the timer through CSL_tmrHwControl().

Enumeration values:

CSL_TMR_CMD_LOAD_PRDLO	Loads the Timer Period Register Low. **Parameters:** *Uint32 **
CSL_TMR_CMD_LOAD_PRDHI	Loads the Timer Period Register High. **Parameters:** *Uint32 **
CSL_TMR_CMD_LOAD_PSCHI	Loads the Timer Pre-scalar value for TIMHI. **Parameters:** *Uint8 **
CSL_TMR_CMD_START_TIMLO	Enable the Timer Low. **Parameters:** *CSL_TmrEnamode **
CSL_TMR_CMD_START_TIMHI	Enable the Timer High. **Parameters:** *CSL_TmrEnamode **
CSL_TMR_CMD_STOP_TIMLO	Stop the Timer Low. **Parameters:** *None*
CSL_TMR_CMD_STOP_TIMHI	Stop the Timer High. **Parameters:** *None*
CSL_TMR_CMD_RESET_TIMLO	Reset the timer Low. **Parameters:** *None*
CSL_TMR_CMD_RESET_TIMHI	Reset the Timer High. **Parameters:** *None*
CSL_TMR_CMD_START64	Start the timer in GPtimer64 OR Chained mode. **Parameters:** *CSL_TmrEnamode **
CSL_TMR_CMD_STOP64	Stop the timer of GPtimer64 OR Chained. **Parameters:** *None*
CSL_TMR_CMD_RESET64	Reset the timer of GPtimer64 OR Chained. **Parameters:** *None*
CSL_TMR_CMD_START_WDT	Enable the timer in watchdog mode. **Parameters:** *CSL_TmrEnamode **

CSL_TMR_CMD_LOAD_WDKEY	Loads the watchdog key. **Parameters:** *Uint16 **

15.4.2 CSL_TmrHwStatusQuery

enum CSL_TmrHwStatusQuery
This enum describes the commands used to get status of various parameters of the Timer. These values are used in CSL_tmrGetHwStatus().

Enumeration values:

CSL_TMR_QUERY_COUNT_LO	Gets the current value of the Timer TIMLO register. **Parameters:** *Uint32 **
CSL_TMR_QUERY_COUNT_HI	Gets the current value of the Timer TIMHI register. **Parameters:** *Uint32 **
CSL_TMR_QUERY_TSTAT_LO	This query command returns the status about whether the TIMLO is running or stopped. **Parameters:** *CSL_TmrTstat **
CSL_TMR_QUERY_TSTAT_HI	This query command returns the status about whether the TIMHI is running or stopped. **Parameters:** *CSL_TmrTstat **
CSL_TMR_QUERY_WDFLAG_STATUS	This query command returns the status about whether the timer is in watchdog mode or not. **Parameters:** *CSL_WdflagBitStatus **

15.4.3 CSL_TmrIpGate

enum CSL_TmrIpGate
This enum describes whether the Timer Clock input is gated or not gated.

Enumeration values:

CSL_TMR_CLOCK_INP_NOGATE	Timer input not gated
CSL_TMR_CLOCK_INP_GATE	Timer input gated

15.4.4 CSL_TmrClksrc

enum CSL_TmrClksrc
This enum describes the Timer Clock source selection.

Enumeration values:

CSL_TMR_CLKSRC_INTERNAL	Timer clock INTERNAL source selection
CSL_TMR_CLKSRC_TMRINP	Timer clock Timer input pin source selection

15.4.5 CSL_TmrEnamode

enum CSL_TmrEnamode
This enum describes the enabling/disabling of Timer.

Enumeration values:

CSL_TMR_ENAMODE_DISABLE	The timer is disabled and maintains current value
CSL_TMR_ENAMODE_ENABLE	The timer is enabled one time
CSL_TMR_ENAMODE_CONT	The timer is enabled continuously

15.4.6 CSL_TmrPulseWidth

enum CSL_TmrPulseWidth
This enum describes the Timer Clock cycles (1/2/3/4).

Enumeration values:

CSL_TMR_PWID_ONECLK	One timer clock cycle
CSL_TMR_PWID_TWOCLKS	Two timer clock cycle
CSL_TMR_PWID_THREECLKS	Three timer clock cycle
CSL_TMR_PWID_FOURCLKS	Four timer clock cycle

15.4.7 CSL_TmrClockPulse

enum CSL_TmrClockPulse
This enum describes the mode of Timer Clock (Pulse/Clock).

Enumeration values:

CSL_TMR_CP_PULSE	Pulse mode
CSL_TMR_CP_CLOCK	Clock mode

15.4.8 CSL_TmrInvInp

enum CSL_TmrInvInp
This enum describes the Timer input inverter control.

Enumeration values:

CSL_TMR_INVINP_UNINVERTED	Uninverted timer input drives timer
CSL_TMR_INVINP_INVERTED	Inverted timer input drives timer

15.4.9 CSL_TmrInvOutp

enum CSL_TmrInvOutp
This enum describes the Timer output inverter control.

Enumeration values:

CSL_TMR_INVOUTP_UNINVERTED	Uninverted timer output
CSL_TMR_INVOUTP_INVERTED	Inverted timer output

15.4.10 CSL_TmrMode

enum CSL_TmrMode
This enum describes the mode of Timer (GPT/WDT/Chained/Unchained).

Enumeration values:

CSL_TMR_TIMMODE_GPT	The timer is in 64-bit GP timer mode
CSL_TMR_TIMMODE_DUAL_UNCHAINED	The timer is in dual 32-bit timer, unchained mode
CSL_TMR_TIMMODE_WDT	The timer is in 64-bit Watchdog timer mode
CSL_TMR_TIMMODE_DUAL_CHAINED	The timer is in dual 32-bit timer, chained mode

15.4.11 CSL_TmrState

enum CSL_TmrState
This enum describes the reset condition of Timer (ON/OFF).

Enumeration values:

CSL_TMR_TIMxxRS_RESET_ON	Timer TIMxx is in reset
CSL_TMR_TIMxxRS_RESET_OFF	Timer TIMHI is not in reset. TIMHI can be used as a 32-bit timer

15.4.12 CSL_TmrTstat

enum CSL_TmrTstat
This enum describes the status of Timer.

Enumeration values:

CSL_TMR_TSTAT_HIGH	Timer status drives High
CSL_TMR_TSTAT_LOW	Timer status drives Low

15.4.13 CSL_TmrWdflagBitStatus

enum CSL_TmrWdflagBitStatus
This enumeration describes the flag bit status of the timer in watchdog mode.

Enumeration values:

CSL_TMR_WDFLAG_NOTIMEOUT	No watchdog timeout occurred
CSL_TMR_WDFLAG_TIMEOUT	Watchdog timeout occurred

15.5 Macros

**#define CSL_TMR_CONFIG_DEFAULTS **

```
{ \
    CSL_TMR_TIMLO_RESETVAL, \
    CSL_TMR_TIMHI_RESETVAL, \
    CSL_TMR_PRDLO_RESETVAL, \
    CSL_TMR_PRDHI_RESETVAL, \
    CSL_TMR_TCR_RESETVAL, \
    CSL_TMR_TGCR_RESETVAL, \
    CSL_TMR_WDTCR_RESETVAL \
}
```

Default values for Config structure.

**#define CSL_TMR_HWSETUP_DEFAULTS **

```
{ \
    CSL_TMR_PRDLO_RESETVAL, \
    CSL_TMR_PRDHI_RESETVAL, \
    CSL_TMR_TIMLO_RESETVAL, \
    CSL_TMR_TIMHI_RESETVAL, \
    (CSL_TmrPulseWidth)CSL_TMR_TCR_PWID_HI_RESETVAL, \
    (CSL_TmrClockPulse)CSL_TMR_TCR_CP_HI_RESETVAL, \
    (CSL_TmrInvOutp)CSL_TMR_TCR_INVOUTP_HI_RESETVAL, \
    (CSL_TmrIpGate)CSL_TMR_TCR_TIEN_LO_RESETVAL, \
    (CSL_TmrClksrc)CSL_TMR_TCR_CLKSRC_LO_RESETVAL, \
    (CSL_TmrPulseWidth)CSL_TMR_TCR_PWID_LO_RESETVAL, \
    (CSL_TmrClockPulse)CSL_TMR_TCR_CP_LO_RESETVAL, \
    (CSL_TmrInvInp)CSL_TMR_TCR_INVINP_LO_RESETVAL, \
    (CSL_TmrInvOutp)CSL_TMR_TCR_INVOUTP_LO_RESETVAL, \
    CSL_TMR_TGCR_PSCHI_RESETVAL, \
    (CSL_TmrMode)CSL_TMR_TGCR_TIMMODE_RESETVAL \
}
```

Default values for hardware setup parameters.

15.6 Typedefs

typedef CSL_TmrObj * CSL_TmrHandle
This data type is used to return the handle to the CSL of the Timer.

Chapter 16
UTOPIA2 Module

Topics

16.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within UTOPIA2 module.

The Universal Test and Operations PHY Interface for ATM (UTOPIA) peripheral is a 50 MHz, 8-Bit Slave-only interface. The UTOPIA is more simplistic than the Ethernet MAC, in that the UTOPIA is serviced directly by the EDMA. The UTOPIA peripheral contains two, two-cell FIFOs, one for transmit and one for receive, with which to buffer up data sent/received across the pins. There is a transmit and a receive event to the EDMA to enable servicing.

The UTOPIA is an ATM controller (ATMC) slave device that interfaces to a master ATM controller.

The UTOPIA slave interface relies on the master ATM controller to provide the necessary control signals such as the clock, enable and address values. Only cell-level handshaking is supported. Both the CPU and enhanced DMA (EDMA) controller can service the UTOPIA.

The UTOPIA slave consists of the transmit interface and the receive interface.

16.2 Functions

This section lists the functions available in the UTOPIA2 module.

16.2.1 UTOPIA2_reset

CSLAPI void UTOPIA2_reset (void)

Description
This function resets UTOPIA2 Control Register and sets the Clock Detect Register.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
UTOPIA2 registers

Example

```
...
UTOPIA2_reset();
...
```

16.2.2 UTOPIA2_getXmtAddr

IDEF Uint32 UTOPIA2_getXmtAddr (void)

Description
This function is to get the transmit address of UTOPIA2. This address is needed to write to the Transmit Port.

Arguments
None

Return Value
`Uint32`

- Address of transmit queue

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 utopXmtAddr;
...
utopXmtAddr = UTOPIA2_getXmtAddr();
...
```

16.2.3 UTOPIA2_getRcvAddr

IDEF Uint32 UTOPIA2_getRcvAddr (void)

Description
This function is to get the receive address of UTOPIA2. This address is required to read from the Receiver Port.

Arguments
None

Return Value
`Uint32`

- Address of receive queue

Pre Condition
None

Post Condition
None

Modifies
None
Example

```
Uint32 utopRcvAddr;
...
utopRcvAddr = UTOPIA2_getRcvAddr();
...
```

16.2.4 UTOPIA2_getEventId

IDEF Uint32 UTOPIA2_getEventId (void)

Description
This function is to get the event Id associated to the UTOPIA2 CPU-interrupt Id.

Arguments
None

Return Value
`Uint32`

- Event Id of UTOPIA2

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 utopEventId;
utopEventId = UTOPIA2_getEventId();
...
```

16.2.5 UTOPIA2_read

IDEF Uint32 UTOPIA2_read (void)

Description
Reads data from the receive queue of UTOPIA2.

Arguments
None

Return Value
`Uint32`

- Data from the receive queue

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 utopRxData;
utopRxData = UTOPIA2_read();
...
```

16.2.6 UTOPIA2_write

IDEF void UTOPIA2_write (Uint32 *val*)

Description
Writes data into the transmit queue of UTOPIA2.

Arguments

```
val        Value to be written into transmit queue
```

Return Value
None

Pre Condition
None

Post Condition
Value passed is written at transmit address of UTOPIA2 i.e. UTOPIA2_XMTQ_ADDR

Modifies
None

Example

```
Uint32 utopTxData = 0x1111FFFF;
...
UTOPIA2_write(utopTxData);
...
```

16.2.7 UTOPIA2_enableXmt

IDEF void UTOPIA2_enableXmt (void)

Description
This function enables transmitter port of UTOPIA2.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Modifies the UXEN bit of UCR register.

Example

```
/* Configure UTOPIA2 */
UTOPIA2_configArgs(0x00040004, /* ucr */
                   0x00FF00FF /* cdr */);
.....
.....
```

```
/* Enables Transmitter port */
UTOPIA2_enableXmt();
...
```

16.2.8 UTOPIA2_enableRcv

IDEF void UTOPIA2_enableRcv (void)

Description
This function enables the receiver port of UTOPIA2

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Modifies the UREN bit of UCR register.

Example

```
/* Configure UTOPIA2 */
UTOPIA2_configArgs(0x00040004, /* ucr */
                   0x00FF00FF /* cdr */
                  );
...
/* Enables Receiver port */
UTOPIA2_enableRcv();
...
```

16.2.9 UTOPIA2_errDisable

IDEF void UTOPIA2_errDisable (Uint32 *errNum*)

Description
This function disables the error interrupt event.

Arguments

```
errNum    Error interrupt event number to be disabled
```

The following are the possible errors from EIPR
- UTOPIA2_ERR_RQS
- UTOPIA2_ERR_RCF
- UTOPIA2_ERR_RCP
- UTOPIA2_ERR_XQS
- UTOPIA2_ERR_XCF

- UTOPIA2_ERR_XCP

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Disables the given error number of EIER register.

Example

```
...
/* Disables the transmit clock fail error bit */
UTOPIA2_errDisable(UTOPIA2_ERR_XCF);
...
```

16.2.10 UTOPIA2_errEnable

IDEF void UTOPIA2_errEnable (Uint32 *errNum*)

Description
Enables the bit of given error condition ID of EIPR.

Arguments

```
errNum      Error interrupt event number to be enabled
```

The following are the possible errors from EIPR
- UTOPIA2_ERR_RQS
- UTOPIA2_ERR_RCF
- UTOPIA2_ERR_RCP
- UTOPIA2_ERR_XQS
- UTOPIA2_ERR_XCF
- UTOPIA2_ERR_XCP

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Sets the given error number of EIER register.

Example

```
...
```

```
/* Enables the transmit clock fail error bit */
UTOPIA2_errEnable(UTOPIA2_ERR_XCF);
...
```

16.2.11 UTOPIA2_errClear

IDEF void UTOPIA2_errClear (Uint32 *errNum*)

Description
This function clears the bit of given error condition ID of EIPR.

Arguments

```
errNum        Error interrupt event number to be cleared
```

The following are the possible errors from EIPR
- UTOPIA2_ERR_RQS
- UTOPIA2_ERR_RCF
- UTOPIA2_ERR_RCP
- UTOPIA2_ERR_XQS
- UTOPIA2_ERR_XCF
- UTOPIA2_ERR_XCP

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Clears the given error number of EIPR register.

Example

```
...
/* Clears the transmit clock fail error bit. */
UTOPIA2_errClear(UTOPIA2_ERR_XCF);
...
```

16.2.12 UTOPIA2_errTest

IDEF Uint32 UTOPIA2_errTest (Uint32 *errNum*)

Description
This function checks the error status of given error number.

Arguments

```
errNum       Error interrupt event number
```

The following are the possible errors from EIPR

- UTOPIA2_ERR_RQS
- UTOPIA2_ERR_RCF
- UTOPIA2_ERR_RCP
- UTOPIA2_ERR_XQS
- UTOPIA2_ERR_XCF
- UTOPIA2_ERR_XCP

Return Value
`Uint32`

- Value of `error interrupt event for specified error event.`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
/* Checking for the transmit clock fail error bit. */
Uint32 errDetect;
UTOPIA2_errEnable(UTOPIA2_ERR_RCF);
errDetect = UTOPIA2_errTest(UTOPIA2_ERR_RCF);
...
```

16.2.13 UTOPIA2_errReset

IDEF void UTOPIA2_errReset (Uint32 *errNum*)

Description
Disables and clears the error interrupt bit associated to the given error number.

Arguments

```
errNum      Error interrupt event number
```

The following are the possible errors from EIPR
- UTOPIA2_ERR_RQS
- UTOPIA2_ERR_RCF
- UTOPIA2_ERR_RCP
- UTOPIA2_ERR_XQS
- UTOPIA2_ERR_XCF
- UTOPIA2_ERR_XCP

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Clears the specified error interrupt event bit of EIPR register.

Example

```
...
/* Disables and clears the transmit clock fail error bit. */
UTOPIA2_errReset(UTOPIA2_ERR_XCF);
...
```

16.2.14 UTOPIA2_config

IDEF void UTOPIA2_config (UTOPIA2_Config * *config*)

Description
Sets up configuration to use the UTOPIA2. The values are set to the UTOPIA2 register (UCR, CDR).

Arguments

```
config    Pointer to an initialized configuration structure
```

Return Value
None

Pre Condition
None

Post Condition
The registers of the UTOPIA2 configured according to value passed.

Modifies
UCR and CDR registers of UTOPIA2.

Example

```
UTOPIA2_Config utopConfig = {     0x00000000, /* ucr */
                                  0x00FF00FF  /* cdr */ };
UTOPIA2_config(&utopConfig);
...
```

16.2.15 UTOPIA2_configArgs

IDEF void UTOPIA2_configArgs (Uint32 *ucr*, Uint32 *cdr*)

Description
This function sets up the UTOPIA2 mode by writing the registers that is passed in.

Arguments

```
ucr          Utopia2 Control Register value

cdr          Clock Detect Register value
```

Return Value
None

Pre Condition
None

Post Condition
The registers of the UTOPIA2 configured according to value passed.

Modifies
UCR and CDR registers of UTOPIA2

Example
```
...
UTOPIA2_configArgs(   0x00000000, /* ucr */
                      0x00FF00FF  /* cdr */);
...
```

16.2.16 UTOPIA2_getConfig

IDEF void UTOPIA2_getConfig (UTOPIA2_Config * ***config*** **)**

Description
Reads the configuration values into the config structure.

Arguments

```
config     Pointer to a configuration structure.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
UTOPIA2_config utopConfig;
UTOPIA2_getConfig(&utopConfig);
...
```

16.3 Data Structures

This section lists the data structures available in the UTOPIA2 module.

16.3.1 UTOPIA2_Config

Detailed Description

The Config structure.
Used to configure the UTOPIA2 using utop_config(ucr,cdr);

Field Documentation

Uint32 UTOPIA2_Config::cdr

Clock Detect Register of UTOPIA2

Uint32 UTOPIA2_Config::ucr

UTOPIA2 Control Register

16.4 Macros

#define UTOPIA2_ERR_RCF 1
Receive clock failed interrupt enable bit

#define UTOPIA2_ERR_RCP 2
Receive clock present interrupt enable bit

#define UTOPIA2_ERR_RQS 0
Receive queue stall interrupt enable bit

#define UTOPIA2_ERR_XCF 17
Transmit clock failed interrupt enable bit

#define UTOPIA2_ERR_XCP 18
Transmit clock present interrupt enable bit

#define UTOPIA2_ERR_XQS 16
Transmit queue stall interrupt enable bit

#define UTOPIA2_INT_RQ 16
Interrupt for Receive queue

#define UTOPIA2_INT_XQ 0
Interrupt for Transmit queue

#define UTOPIA2_RCVQ_ADDR CSL_UTOPIA2_RX_EDMA_REGS
Base address of the UTOPIA2 receive queue

#define UTOPIA2_XMTQ_ADDR CSL_UTOPIA2_TX_EDMA_REGS
Base address of the UTOPIA2 transmit queue

Chapter 17
VCP2 Module

Topics

17.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within VCP2 module.

Viterbi Decoder Coprocessor 2 (VCP2) is a programmable peripheral for decoding of convolutional codes. The VCP2 is controlled via memory mapped control registers and data buffers. The VCP2 can be used for channel decoding of voice and low bit-rate data channels found in third generation cellular standards that require decoding of convolutional encoded data. The VCP coprocessor has been designed to perform forward error correction for 2G and 3G wireless systems. The VCP can support 1941 12.2 Kbps class A 3G voice channels running at 333 Mhz.

The VCP2 supports:

- Unlimited frame sizes
- Code rates 1/2, 1/3, and 1/4
- Constraint lengths 5, 6, 7, 8, and 9
- Programmable encoder polynomials
- Programmable reliability and convergence lengths
- Hard and soft decoded decisions
- Tail and convergent modes
- Yamamoto logic
- Tail biting logic
- Various input and output FIFO lengths

17.2 Functions

This section lists the functions available in the VCP2 module.

17.2.1 VCP2_genParams

void VCP2_genParams (VCP2_BaseParams *restrict *pConfigBase*, VCP2_Params *restrict *pConfigParams*)

Description
This function calculates the VCP parameters based on the input VCP2_BaseParams object values and sets the values to the output VCP2_Params parameters structure.

Arguments

```
pConfigBase     Pointer to VCP base parameters structure.

pConfigParams   Pointer to output VCP channel parameters structure.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Input VCP2_Params structure instance pointed by pConfigParams.

Example

```
VCP2_Params       vcpParam;
VCP2_BaseParams   vcpBaseParam;

vcpBaseParam.rate            =   VCP2_RATE_1_4;
vcpBaseParam.constLen        =   5;
vcpBaseParam.frameLen        =   2042;
vcpBaseParam.yamTh           =   50;
vcpBaseParam.stateNum        =   63;
vcpBaseParam.tbConvrgMode    =   FALSE;
vcpBaseParam.decision        =   VCP2_DECISION_HARD;
vcpBaseParam.readFlag        =   VCP2_OUTF_YES;
vcpBaseParam.tailBitEnable   =   FALSE;
vcpBaseParam.traceBackIndex =   0x0;
vcpBaseParam.outOrder        =   VCP2_OUTORDER_0_31;
vcpBaseParam.perf            =   VCP2_SPEED_CRITICAL;
VCP2_genParams(&vcpBaseParam, &vcpParam);
...
```

17.2.2 VCP2_genIc

void VCP2_genIc **(** **[VCP2_Params]() *restrict** ***pConfigParams*,**
[VCP2_ConfigIc]() *restrict ***pConfigIc***
)

Description
This function generates the required input configuration register values needed to program the VCP based on the parameters provided by VCP2_Params object values.

Arguments

```
pConfigParams   Pointer to channel parameters structure

pConfigIc       Pointer to input configuration structure
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Input VCP2_ConfigIc structure instance pointed by pConfigIc.

Example

```
VCP2_Params       vcpParam;
VCP2_BaseParams   vcpBaseParam;
VCP2_ConfigIc     vcpConfig;

...
vcpBaseParam.rate           =    VCP2_RATE_1_4;
vcpBaseParam.constLen       =    5;
vcpBaseParam.frameLen       =    2042;
vcpBaseParam.yamTh          =    50;
vcpBaseParam.stateNum       =    63;
vcpBaseParam.tbConvrgMode   =    FALSE;
vcpBaseParam.decision       =    VCP2_DECISION_HARD;
vcpBaseParam.readFlag       =    VCP2_OUTF_YES;
vcpBaseParam.tailBitEnable  =    FALSE;
vcpBaseParam.traceBackIndex =    0x0;
vcpBaseParam.outOrder       =    VCP2_OUTORDER_0_31;
vcpBaseParam.perf           =    VCP2_SPEED_CRITICAL;
...
VCP2_genParams (&vcpBaseParam, &vcpParam);
VCP2_genIc (&vcpParam, &vcpConfig);
...
```

17.2.3 VCP2_ceil

CSL_IDEF_INLINE Uint32 VCP2_ceil (Uint32 *val*, Uint32 *pwr2*)

Description
Returns the value rounded to the nearest integer, greater than or equal to (val/(2^pwr2)).

Arguments

```
val         Value to be augmented.

pwr2        The power of two by which val must be divisible.
```

Return Value
`Uint32`

- Value - The smallest number which when multiplied by 2^pwr2 is greater than val.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32     val1 = 512;
Uint32     val2 = 4;
Uint32     val3;

val3 = VCP2_ceil(val1, val2);
...
```

17.2.4 VCP2_normalCeil

CSL_IDEF_INLINE Uint32 VCP2_normalCeil (Uint32 *val1*, Uint32 *val2*)

Description
Returns the value rounded to the nearest integer, greater than or equal to (val1/val2)

Arguments

```
val1        Value to be augmented.
```

```
val2        Value by which val1 must be divisible.
```

Return Value
`Uint32`

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 bmCnt = 512;
Uint32 numBmFrames;

// Number of frame transfers with number of bytes
// transferred to the VCP2 per receive event – 128

numBmFrames = VCP2_normalCeil(bmCnt, 128);
...
```

17.2.5 VCP2_getBmEndian

CSL_IDEF_INLINE Uint32 VCP2_getBmEndian (void)

Description
This function returns the value programmed into the VCPEND register for the branch metrics data for Big Endian mode indicating whether the data is in its native 8-bit format ('1') or 32-bit word packed ('0').

Arguments
None

Return Value
`Uint32`

- Value - Branch metric memory format.

  0 - 32-bit word packed.

  1 - Native (8 bits).

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
if (VCP2_getBmEndian())
{
    ...
} /* end if */
...
```

17.2.6 VCP2_getIcConfig

CSL_IDEF_INLINE void VCP2_getIcConfig (VCP2_ConfigIc * *configIc*)

Description
This function gets the current VCPIC register values and puts them in a structure of type VCP2_ConfigIc.

Arguments

```
configIc    Pointer to the structure of type VCP2_ConfigIc to
            hold the values of VCPIC registers.
```

Return Value
None

Pre Condition
None

Post Condition
The structure of type VCP2_ConfigIc passed as argument contains the values of the VCP configuration registers.

Modifies
Input structure of type VCP2_ConfigIc.

Example

```
VCP2_ConfigIc    configIc;

VCP2_getIcConfig(&configIc);
...
```

17.2.7 VCP2_getNumInFifo

CSL_IDEF_INLINE Uint32 VCP2_getNumInFifo (void)

Description
This function returns the count, number of symbols currently in the input FIFO.

Arguments
None

Return Value
`Uint32`

- Value - Number of symbols in the branch metric input FIFO buffer.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 numSym;
numSym = VCP2_getNumInFifo();
...
```

17.2.8 VCP2_getNumOutFifo

CSL_IDEF_INLINE Uint32 VCP2_getNumOutFifo (void)

Description
This function returns the count, number of symbols currently in the output FIFO.

Arguments
None

Return Value
`Uint32`

- Value - Number of symbols present in the output FIFO buffer.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32  numSym;
numSym = VCP2_getNumOutFifo();
...
```

17.2.9 VCP2_getSdEndian

CSL_IDEF_INLINE Uint32 VCP2_getSdEndian (void)

Description
This function returns the value programmed into the VCPEND register for the soft decision data for Big Endian mode indicating whether the data is in its native 8-bit format ('1') or 32-bit word packed ('0').

Arguments
None

Return Value
`Uint32`

- Value - Soft decisions memory format.

  0 - 32-bit word packed.

  1 - Native (8 bits).

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
if (VCP2_getSdEndian ())
{
    ...
} /* end if */
...
```

17.2.10 VCP2_getStateIndex

CSL_IDEF_INLINE Uint8 VCP2_getStateIndex (void)

Description
This function returns an index for the final maximum state metric.

Arguments
None

Return Value
`Uint8`

- Value - Final maximum state metric index.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint8   index;
...
index = VCP2_getStateIndex();
...
```

17.2.11 VCP2_getYamBit

CSL_IDEF_INLINE Uint8 VCP2_getYamBit (void)

Description
This function returns the value of the Yamamoto bit after the VCP decoding. This bit is a quality indicator and is only used if the yamamoto logic is enabled.

Arguments
None

Return Value
`Uint8`

- Value - Yamamoto bit result.

  0 - at least one trellis stage had an absolute difference less than the Yamamoto threshold and the decoded frame has poor quality.
  1 - no trellis stage had an absolute difference less than the Yamamoto threshold and the frame has good quality.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint8   yamBit;

yamBit = VCP2_getYamBit();
...
```

17.2.12 VCP2_getMaxSm

CSL_IDEF_INLINE Int16 VCP2_getMaxSm (void)

Description
This function returns the final maximum state metric after the VCP has completed its decoding.

Arguments
None

Return Value
`Int16`

- Value - Maximum state metric value for the final trellis stage.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Int16   maxSm;

maxSm = VCP2_getMaxSm();
...
```

17.2.13 VCP2_getMinSm

CSL_IDEF_INLINE Int16 VCP2_getMinSm (void)

Description
This function returns the final minimum state metric after the VCP has completed its decoding.

Arguments
None

Return Value
`Int16`

- Value - Minimum state metric value for the final trellis stage.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Int16   minSm;
minSm = VCP2_getMinSm();
...
```

17.2.14 VCP2_icConfig

CSL_IDEF_INLINE void VCP2_icConfig (VCP2_ConfigIc * *vcpConfigIc*)

Description
This function programs the VCP input configuration registers with the values provided through the VCP2_ConfigIc structure.

Arguments

```
vcpConfigIc     Pointer to VCP2_ConfigIc structure instance
                containing the input configuration register values.
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP input configuration registers.

Example

```
VCP2_ConfigIc     configIc;
configIc.ic0  =  0xf0b07050;
configIc.ic1  =  0x10320000;
configIc.ic2  =  0x000007fa;
configIc.ic3  =  0x00000054;
configIc.ic4  =  0x00800800;
configIc.ic5  =  0x51f3000c;
VCP2_icConfig(&configIc);
...
```

17.2.15 VCP2_icConfigArgs

CSL_IDEF_INLINE void VCP2_icConfigArgs (Uint32 *ic0*,
Uint32 *ic1*,
Uint32 *ic2*,
Uint32 *ic3*,
Uint32 *ic4*,

Uint32 ***ic5***

)

Description
This function programs the VCP input configuration registers with the given values.

Arguments

`ic0`	`Value to program input configuration register 0`
`ic1`	`Value to program input configuration register 1`
`ic2`	`Value to program input configuration register 2`
`ic3`	`Value to program input configuration register 3`
`ic4`	`Value to program input configuration register 4`
`ic5`	`Value to program input configuration register 5`

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP input configuration registers.

Example

```
Uint32 ic0, ic1, ic2, ic3, ic4, ic5;
...
ic0  =  0xf0b07050;
ic1  =  0x10320000;
ic2  =  0x000007fa;
ic3  =  0x00000054;
ic4  =  0x00800800;
ic5  =  0x51f3000c;
VCP2_icConfigArgs(ic0, ic1, ic2, ic3, ic4, ic5);
...
```

17.2.16 VCP2_setBmEndian

CSL_IDEF_INLINE void VCP2_setBmEndian **(** **Uint32** ***bmEnd*** **)**

Description
This function programs the VCP to view the format of the branch metrics data as either native 8-bit format ('1') or values packed into 32-bit words in little endian format ('0').

Arguments

```
bmEnd   '1' for native 8-bit format and '0' for 32-bit word packed
        format
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
Uint32 bmEnd = VCP2_END_NATIVE;
VCP2_setBmEndian(bmEnd);
...
```

17.2.17 VCP2_setNativeEndian

CSL_IDEF_INLINE void VCP2_setNativeEndian (void)

Description
This function programs the VCP to view the format of all data as native 8-bit format.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
VCP2_setNativeEndian();
...
```

17.2.18 VCP2_setPacked32Endian

CSL_IDEF_INLINE void VCP2_setPacked32Endian (void)

Description
This function programs the VCP to view the format of all data as packed data in 32-bit words.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
VCP2_setPacked32Endian();
...
```

17.2.19 VCP2_setSdEndian

CSL_IDEF_INLINE void VCP2_setSdEndian (Uint32 *sdEnd*)

Description
This function programs the VCP to view the format of the soft decision data as either native 8-bit format ('1') or values packed into 32-bit words in little endian format ('0').

Arguments

```
sdEnd   '1' for native 8-bit format and '0' for 32-bit word packed
        format
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
Uint32 sdEnd = VCP2_END_NATIVE;
VCP2_setSdEndian(sdEnd);
...
```

17.2.20 VCP2_addPoly

CSL_IDEF_INLINE void VCP2_addPoly (VCP2_Poly * *poly*, VCP2_Params * *params*)

Description
This function is used to add either predefined or user defined polynomials to the generated VCP2_Params.

Arguments

poly	Pointer to the structure of type VCP2_Poly containing the values of generator polynomials.
params	Pointer to the structure of type VCP2_Params containing the generated values for input configuration registers.

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
Structure of type VCP2_Params passed as argument to the function.

Example

```
VCP2_Poly  poly = {VCP2_GEN_POLY_3, VCP2_GEN_POLY_1,
                   VCP2_GEN_POLY_2, VCP2_GEN_POLY_3};
VCP2_Params      params;
VCP2_BaseParams  baseParams;
...
VCP2_genParams(&baseParams, &params);
VCP2_addPoly(&poly, &params);
...
```

17.2.21 VCP2_statError

CSL_IDEF_INLINE Bool VCP2_statError (void)

Description
This function returns a boolean value indicating whether any VCP error has occurred.

Arguments
None

Return Value
Bool

- bitStatus - ERR bit field value of VCP status register 0.

  0 – No error.

  1 – VCP paused due to error.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
VCP2_Errors error;
/* check whether an error has occurred */
if (VCP2_statError()) {
    VCP2_getErrors(&error);
} /* end if */
...
```

17.2.22 VCP2_statInFifo

CSL_IDEF_INLINE Uint32 VCP2_statInFifo (void)

Description
This function returns the input FIFO's empty status flag. A '1' indicates that the input FIFO is empty and a '0' indicates it is not empty.

Arguments
None

Return Value
`Uint32`

- bitStatus - IFEMP bit field value of VCP status register 0.

  0 – Input FIFO is not empty.

  1 – Input FIFO is empty.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (VCP2_statInFifo()) {
    ...
} /* end if */
...
```

17.2.23 VCP2_statOutFifo

CSL_IDEF_INLINE Uint32 VCP2_statOutFifo (void)

Description
This function returns the output FIFO's full status flag. A '1' indicates that the output FIFO is full and a '0' indicates it is not full.

Arguments
None

Return Value
`Uint32`

- bitStatus - OFFUL bit field value of VCP status register 0.

  0 – Output FIFO is not full.

  1 – Output FIFO is full.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (VCP2_statOutFifo()) {
    ...
} /* end if */
...
```

17.2.24 VCP2_statPause

CSL_IDEF_INLINE Uint32 VCP2_statPause (void)

Description
This function returns the PAUSE bit status indicating whether the VCP is paused or not.

Arguments
None

Return Value
`Uint32`

- bitStatus - PAUSE bit field value of VCP status register 0.

  0 – VCP is not paused.

  1 – VCP is paused.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
/* Pause the VCP */
VCP2_pause ();
/* Wait for pause to take place */
while (!VCP2_statPause());
...
```

17.2.25 VCP2_statRun

CSL_IDEF_INLINE Uint32 VCP2_statRun (void)

Description
This function returns the RUN bit status indicating whether the VCP is running or not.

Arguments
None

Return Value
`Uint32`

- bitStatus - RUN bit field value of VCP status register 0.

  0 – VCP is not running.

  1 – VCP is running.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
/* start the VCP */
VCP2_start ();
/* check that the VCP is running */
while (!VCP2_statRun());
...
```

17.2.26 VCP2_statSymProc

CSL_IDEF_INLINE Uint32 VCP2_statSymProc (void)

Description
This function returns the number of symbols processed, NSYMPROC bitfield of VCP.

Arguments
None

Return Value
`Uint32`

- Value - Number of symbols processed.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32 numSym;
numSym = VCP2_statSymProc();
...
```

17.2.27 VCP2_statWaitIc

CSL_IDEF_INLINE Uint32 VCP2_statWaitIc (void)

Description
This function returns the WIC bit status indicating whether the VCP is waiting to receive new input configuration values.

Arguments
None

Return Value
`Uint32`

- bitStatus - WIC bit field value of VCP status register 0.

  0 – VCP is not waiting for input configuration words.

  1 – VCP is waiting for input configuration words.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
if (VCP2_statWaitIc()) {
    ...
} /* end if */
...
```

17.2.28 VCP2_start

CSL_IDEF_INLINE void VCP2_start (void)

Description
This function starts the VCP by writing a start command to the VCPEXE register.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
VCP is started.

Modifies
VCP execution register.

Example

```
...
VCP2_start();
...
```

17.2.29 VCP2_pause

CSL_IDEF_INLINE void VCP2_pause (void)

Description
This function pauses the VCP by writing a pause command to the VCPEXE register.

Arguments
None

Return Value
None

Pre Condition
The VCP should be operating in debug mode.

Post Condition
VCP is paused.

Modifies
VCP execution register

Example

```
...
VCP2_pause();
...
```

17.2.30 VCP2_unpause

CSL_IDEF_INLINE void VCP2_unpause (void)

Description
This function un-pauses the VCP, previously paused by VCP2_pause() function, by writing the un-pause command to the VCPEXE register. This function restarts the VCP at the beginning of current trace back, and VCP will run to normal completion.

Arguments
None

Return Value
None

Pre Condition
The VCP should be operating in debug mode.

Post Condition
VCP is restarted.

Modifies
VCP execution register.

Example

```
...
VCP2_unpause();
...
```

17.2.31 VCP2_stepTraceback

CSL_IDEF_INLINE void VCP2_stepTraceback (void)

Description
This function un-pauses the VCP, previously paused by VCP2_pause() function, by writing the un-pause command to the VCPEXE register. This function restarts the VCP at the beginning of current trace back and halts at the next trace back (i.e. Step Single Trace back).

Arguments
None

Return Value
None

Pre Condition
The VCP should be operating in debug mode.

Post Condition
VCP is restarted.

Modifies
VCP execution register.

Example
```
...
VCP2_stepTraceback();
...
```

17.2.32 VCP2_reset

CSL_IDEF_INLINE void VCP2_reset (void)

Description
This function sets all the VCP control registers to their default values.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
All registers in the VCP are reset except for the execution register, endian register, emulation register and other internal registers.

Modifies
VCP execution register

Example
```
VCP2_reset();
...
```

17.2.33 VCP2_getErrors

CSL_IDEF_INLINE void VCP2_getErrors (VCP2_Errors * *pVcpErr*)

Description
This function will acquire the VCPERR register values and fill in the fields of VCP2_Error structure and pass it back as the results.

Arguments

```
pVcpErr     Pointer to the VCP2_Errors structure instance.
```

Return Value
None

Pre Condition
None

Post Condition
The fields of the VCP2_Errors structure indicate the respective errors, if occurred.

Modifies
VCPSTAT0 register as a side effect. Clears ERR bit of VCPSTAT0 register.

Example

```
VCP2_Errors error;
...
/* Check whether an error has occurred */
if (VCP2_statError()) {
    VCP2_getErrors(&error);
} /* end if */
...
```

17.2.34 VCP2_statEmuHalt

CSL_IDEF_INLINE Uint32 VCP2_statEmuHalt (void)

Description
This function returns the EMUHALT bit status indicating whether the VCP halt is due to emulation or not.

Arguments
None

Return Value
`Uint32`

- bitStatus - Emuhalt bit field value of VCP status register 0.

  0 – Not halt due to emulation.

  1 – Halt due to emulation.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
if (VCP2_statEmuHalt()) {
    ...
}/* end if */
...
```

17.2.35 VCP2_getVssSleepMode

CSL_IDEF_INLINE Uint32 VCP2_getVssSleepMode (void)

Description
This function returns the value programmed into VCPEND register for sleep mode indicating if sleep mode is disabled or if internal power down control is enabled for SLPZVSS.

Arguments
None

Return Value
`Uint32`

- Value - Sleep mode enable/disable.

  0 - Sleep mode disabled.

  1 - Sleep mode enabled.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32  slpMode;
slpMode = VCP2_getVssSleepMode();
...
```

17.2.36 VCP2_getVddSleepMode

CSL_IDEF_INLINE Uint32 VCP2_getVddSleepMode (void)

Description
This function returns the value programmed into VCPEND register for sleep mode indicating if sleep mode is disabled or if internal power down control is enabled for SLPZVDD.

Arguments
None

Return Value
`Uint32`

- Value - Sleep mode enable/disable.

  0 - Sleep mode disabled.

  1 - Sleep mode enabled.

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32  slpMode;
slpMode = VCP2_getVddSleepMode();
...
```

17.2.37 VCP2_setVssSleepMode

CSL_IDEF_INLINE void VCP2_setVssSleepMode (Uint32 *slpMode*)

Description
This function either disables sleep mode or enables the internal power down control of SLPZVSS.

Arguments

`slpMode`	`'0' to disable sleep mode and '1' to enable internal power down control for SLPZVSS.`

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
...
VCP2_setVssSleepMode(1);
...
```

17.2.38 VCP2_setVddSleepMode

CSL_IDEF_INLINE void VCP2_setVddSleepMode (Uint32 *slpMode*)

Description
This function either disables sleep mode or enables the internal power down control of SLPZVDD.

Arguments

slpMode	'0' to disable sleep mode and '1' to enable internal power down control for SLPZVDD.

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP endian register

Example

```
...
VCP2_setVddSleepMode(1);
...
```

17.2.39 VCP2_emuEnable

CSL_IDEF_INLINE void VCP2_emuEnable (Uint16 *emuMode*)

Description
This function enables the emulation/debug mode of VCP.

Arguments

emuMode	'0' to halt VCP at the end of completion of the current window of state metric processing or at the end of a frame.

```
'1' to halt the VCP at the end of completion of the
processing of the frame.
```

Return Value
None

Pre Condition
None

Post Condition
Emulation mode is enabled.

Modifies
VCP emulation control register.

Example

```
...
Uint16  emuMode = VCP2_EMUHALT_DEFAULT;
VCP2_emuEnable(emuMode);
...
```

17.2.40 VCP2_emuDisable

CSL_IDEF_INLINE void VCP2_emuDisable (void)

Description
This function disables the emulation/debug mode of VCP.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
VCP emulation control register

Example

```
...
VCP2_emuDisable();
...
```

17.3 Data Structures

This section lists the data structures available in the VCP2 module.

17.3.1 VCP2_ConfigIc

Detailed Description
VCP input configuration structure that holds all of the configuration values that are to be transferred to the VCP via the EDMA.

Field Documentation

Uint32 VCP2_ConfigIc::ic0
Value of VCP input configuration register 0

Uint32 VCP2_ConfigIc::ic1
Value of VCP input configuration register 1

Uint32 VCP2_ConfigIc::ic2
Value of VCP input configuration register 2

Uint32 VCP2_ConfigIc::ic3
Value of VCP input configuration register 3

Uint32 VCP2_ConfigIc::ic4
Value of VCP input configuration register 4

Uint32 VCP2_ConfigIc::ic5
Value of VCP input configuration register 5

17.3.2 VCP2_Params

Detailed Description
VCP channel parameters structure that holds all of the information concerning the user channel. These values are used to generate the appropriate input configuration values for the VCP.

Field Documentation

Uint8 VCP2_Params::bmBuffLen
Branch metrics buffer length in input FIFO

Uint8 VCP2_Params::constLen
Constraint length

Uint16 VCP2_Params::convDist
Convergence distance

Uint8 VCP2_Params::decBuffLen
Decisions buffer length in output FIFO

Uint8 VCP2_Params::decision
Decision selection: hard or soft

Uint16 VCP2_Params::frameLen
Frame length i.e. number of symbols in a frame

Int16 VCP2_Params::maxSm
Maximum initial state metric

Int16 VCP2_Params::minSm
Minimum initial state metric

Uint16 VCP2_Params::numBmFrames
Number of branch metric frames

Uint16 VCP2_Params::numDecFrames
Number of decision frames

Uint16 VCP2_Params::outOrder
Output data ordering

Uint8 VCP2_Params::poly0
Polynomial 0

Uint8 VCP2_Params::poly1
Polynomial 1

Uint8 VCP2_Params::poly2
Polynomial 2

Uint8 VCP2_Params::poly3
Polynomial 3

VCP2_Rate VCP2_Params::rate
Code rate

Uint8 VCP2_Params::readFlag
Output parameters read flag

Uint16 VCP2_Params::relLen
Reliability length

Uint8 VCP2_Params::stateNum
State index set to the maximum initial state metric

Uint8 VCP2_Params::traceBack
Traceback mode

Bool VCP2_Params::traceBackEn
Traceback state index enable/disable

Uint16 VCP2_Params::traceBackIndex
Traceback state index

Uint16 VCP2_Params::yamTh
Yamamoto threshold value

17.3.3 VCP2_BaseParams

Detailed Description
VCP base parameter structure that is used to configure the VCP parameters structure with the given values using VCP2_genParams() function.

Field Documentation

Uint8 VCP2_BaseParams::constLen
Constraint length

Uint8 VCP2_BaseParams::decision
Output decision type

Uint16 VCP2_BaseParams::frameLen
Frame length

Uint8 VCP2_BaseParams::outOrder
Output data ordering

Uint8 VCP2_BaseParams::perf
Performance and speed

VCP2_Rate VCP2_BaseParams::rate
Code rate

Uint8 VCP2_BaseParams::readFlag
Output parameters read flag

Uint8 VCP2_BaseParams::stateNum
Maximum initial state metric value

Bool VCP2_BaseParams::tailBitEnable
Enable/Disable tail biting

Bool VCP2_BaseParams::tbConvrgMode
Traceback convergement mode

Uint16 VCP2_BaseParams::traceBackIndex
Tailbiting traceback index mode

Uint16 VCP2_BaseParams::yamTh
Yamamoto threshold value

17.3.4 VCP2_Errors

Detailed Description
VCP Error structure

Field Documentation

Bool VCP2_Errors::fctlErr
Reliability + convergence distance error

Bool VCP2_Errors::ftlErr
Frame length error

Bool VCP2_Errors::maxminErr
Max-Min error

Bool VCP2_Errors::symrErr
SYMR error

Bool VCP2_Errors::symxErr
SYMX error

Bool VCP2_Errors::tbnaErr
Traceback mode error

17.3.5 VCP2_Poly

Detailed Description
VCP generator polynomials structure

Field Documentation

Uint8 VCP2_Poly::poly0
Generator polynomial 0

Uint8 VCP2_Poly::poly1
Generator polynomial 1

Uint8 VCP2_Poly::poly2
Generator polynomial 2

Uint8 VCP2_Poly::poly3
Generator polynomial 3

17.4 Macros

#define hVcp2 ((CSL_Vcp2ConfigRegs*)CSL_VCP2_0_REGS)
Handle to access VCP2 registers accessible through config bus

#define hVcp2Vbus ((CSL_Vcp2EdmaRegs *)CSL_VCP2_EDMA_REGS)
Handle to access VCP2 registers accessible through EDMA bus

#define VCP2_DECISION_HARD CSL_VCP2_VCPIC5_SDHD_HARD
Output decision type: Hard decisions

#define VCP2_DECISION_SOFT CSL_VCP2_VCPIC5_SDHD_SOFT
Output decision type: Soft decisions

#define VCP2_EMUHALT_DEFAULT CSL_VCP2_VCPEMU_SOFT_HALT_DEFAULT
EMU mode: VCP halts at the end of completion of the current window of state metric processing or at the end of a frame

#define VCP2_EMUHALT_FRAMEEND CSL_VCP2_VCPEMU_SOFT_HALT_FRAMEEND
EMU mode : VCP halts at the end of completion of the processing of the frame

#define VCP2_END_NATIVE CSL_VCP2_VCPEND_SD_NATIVE
Soft decisions memory format: Native (8 bits)

#define VCP2_END_PACKED32 CSL_VCP2_VCPEND_SD_32BIT
Soft decisions memory format: 32-bit word packed

#define VCP2_GEN_POLY_0 0x30
GSM/Edge/GPRS generator polynomial 0

#define VCP2_GEN_POLY_1 0xB0
GSM/Edge/GPRS generator polynomial 1

#define VCP2_GEN_POLY_2 0x50
GSM/Edge/GPRS generator polynomial 2

#define VCP2_GEN_POLY_3 0xF0
GSM/Edge/GPRS generator polynomial 3

#define VCP2_GEN_POLY_4 0x6C
GSM/Edge/GPRS generator polynomial 4

#define VCP2_GEN_POLY_5 0x94

GSM/Edge/GPRS generator polynomial 5

#define VCP2_GEN_POLY_6 0xF4
GSM/Edge/GPRS generator polynomial 6

#define VCP2_GEN_POLY_7 0xE4
GSM/Edge/GPRS generator polynomial 7

#define VCP2_GEN_POLY_GNULL 0x00
NULL generator polynomial for GSM/Edge/GPRS

#define VCP2_OUTF_NO   CSL_VCP2_VCPIC5_OUTF_NO
Output parameters read flag: VCP output parameters read event is not generated

#define VCP2_OUTF_YES   CSL_VCP2_VCPIC5_OUTF_YES
Output parameters read flag: VCP output parameters read event is generated

#define VCP2_OUTORDER_0_31   CSL_VCP2_VCPIC3_OUT_ORDER_LSB
Out order of VCP output for decoded data: 0 to 31

#define VCP2_OUTORDER_31_0   CSL_VCP2_VCPIC3_OUT_ORDER_MSB
Out order of VCP output for decoded data: 31 to 0

#define VCP2_PERF_CRITICAL   2
Performance critical

#define VCP2_PERF_DEFAULT   VCP2_SPEED_CRITICAL
Default value

#define VCP2_PERF_MOST_CRITICAL   3
Performance most critical

#define VCP2_RATE_1_2   2
Code rate = 2

#define VCP2_RATE_1_3   3
Code rate = 3

#define VCP2_RATE_1_4   4
Code rate = 4

#define VCP2_SPEED_CRITICAL   0
Speed critical

#define VCP2_SPEED_MOST_CRITICAL   1
Speed most critical

#define VCP2_TRACEBACK_CONVERGENT   CSL_VCP2_VCPIC5_TB_CONV
Traceback mode: Convergent

#define VCP2_TRACEBACK_MIXED   CSL_VCP2_VCPIC5_TB_MIX
Traceback mode: Mixed

#define VCP2_TRACEBACK_NONE   CSL_VCP2_VCPIC5_TB_NO
No trace back allowed

#define VCP2_TRACEBACK_TAILED   CSL_VCP2_VCPIC5_TB_TAIL
Traceback mode: Tailed

#define VCP2_UNPAUSE_NORMAL   CSL_VCP2_VCPEXE_COMMAND_RESTART
VCP unpause type: VCP restarts

#define VCP2_UNPAUSE_ONESW   CSL_VCP2_VCPEXE_COMMAND_RESTART_PAUSE
VCP unpause type: VCP restarts and processes one sliding window before pausing again

17.5 Typedefs

typedef Uint32 VCP2_Rate
VCP code rate type

Chapter 18
BWMNGMT Module

Topics

18.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within BWMNGMT module.

The Bandwidth management module used to avoid a requestors (CPU, SDMA, IDMA, and Coherence Operations) being blocked from accessing a resources (L1P, L1D, L2, and configuration bus) for a long period of time.

The following four resources are managed by the BWM control hardware:

- Level 1 Program (L1P) SRAM/Cache
- Level 1 Data (L1D) SRAM/Cache
- Level 2 (L2) SRAM/Cache
- Memory-mapped registers configuration bus

18.2 Functions

This section lists the functions available in the BWMNGMT module.

18.2.1 CSL_bwmngmtInit

CSL_Status CSL_bwmngmtInit (CSL_BwmngmtContext * *pContext*)

Description
This is the initialization function for the BWMNGMT CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext   Context information for the instance. Should be NULL
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
CSL_Status          status;
...
status = CSL_bwmngmtInit(NULL);
...
```

18.2.2 CSL_bwmngmtOpen

CSL_BwmngmtHandle CSL_bwmngmtOpen (CSL_BwmngmtObj * *pBwmngmtObj*, CSL_InstNum *bwmngmtNum*, CSL_BwmngmtParam * *pBwmngmtParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the instance and returns a handle to the BWMNGMT instance. The open call sets up the data structures for the particular instance of BWMNGMT device. The device can be re-opened anytime after it has been normally closed, if so

required. The handle returned by this call is input as an essential argument for rest of the APIs described for this module.

Arguments

pBwmngmtObj	Pointer to the BWMNGMT instance object
bwmngmtNum	Instance of the BWMNGMT to be opened
pBwmngmtParam	Pointer to module specific parameters
pStatus	Pointer for returning status of the function call

Return Value
CSL_BwmngmtHandle

- Valid BWMNGMT instance handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The BWMNGMT module must be successfully initialized via CSL_bwmngmtInit() before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- CSL_SOK - Valid BWMNGMT handle is returned.
- CSL_ESYS_FAIL - The BWMNGMT instance is invalid.
- CSL_ESYS_INVPARAMS – The Obj structure passed is invalid.

2. BWMNGMT object structure is populated.

Modifies
1. The status variable
2. BWMNGMT object structure

Example
```
CSL_Status            status;
CSL_BwmngmtObj        bwmngmtObj;
CSL_BwmngmtHandle     hBwmngmt;

CSL_bwmngmtInit(NULL);
hBwmngmt = CSL_bwmngmtOpen(&bwmngmtObj,
                           CSL_BWMNGMT,
                           NULL,
                           &status
                           );
...
```

18.2.3 CSL_bwmngmtClose

CSL_Status CSL_bwmngmtClose (CSL_BwmngmtHandle *hBwmngmt*)

Description
This function closes the specified instance of BWMNGMT.

Arguments

`hBwmngmt`	`Handle to the BWMNGMT instance`

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_bwmngmtInit() and CSL_bwmngmtOpen() must be called successfully in order before calling CSL_bwmngmtClose().

Post Condition
The BWMNGMT CSL APIs can not be called until the BWMNGMT CSL is reopened again using CSL_bwmngmtOpen().

Modifies
CSL_bwmngmtObj structure instance values

Example
```
CSL_BwmngmtHandle     hBwmngmt;
CSL_Status            status;
...

status = CSL_bwmngmtClose(hBwmngmt);
...
```

18.2.4 CSL_bwmngmtHwSetup

CSL_Status CSL_bwmngmtHwSetup (CSL_BwmngmtHandle *hBwmngmt*, CSL_BwmngmtHwSetup * *setup*)

Description
Configures the BWMNGMT using the values passed in through the setup structure. For information passed through the HwSetup Data structure, refer CSL_BwmngmtHwSetup.

Arguments

`hBwmngmt`	`Handle to the BWMNGMT instance`
`setup`	`Setup structure for BWMNGMT`

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - If setup is NULL

Pre Condition
Both CSL_bwmngmtInit() and CSL_bwmngmtOpen() must be called successfully in order before calling this function. The main setup structure consists of fields used for the configuration at start up. The user must allocate space for it and fill in the main setup structure fields appropriately before a call to this function is made.

Post Condition
BWMNGMT registers are configured according to the hardware setup parameters.

Modifies
The following registers and fields are programmed by this API
1. CPU Arbitration Parameters
   - PRI field set in L1D, L2 and/or EXT
   - MAXWAIT field set in L1D, L2 and/or EXT

2. IDMA Arbitration Parameter
   - MAXWAIT field set in L1D, L2 and/or EXT

3. SLAP Arbitration Parameter
   - MAXWAIT field set in L1D, L2 and/or EXT

4. MAP Arbitration Parameter
   - PRI field set in EXT

5. UC Arbitration Parameter
   - MAXWAIT field set in L1D and/or L2

control: bitmask indicates which of the three control blocks (L1D, L2 and EXT) will be set with the associated PRI and MAXWAIT values.

Note That if associated control block is not programmable for given requestor then it will not ignored but no error will be provide. This allows the user to set control to CSL_BWMNGMT_BLOCK_ALL which is the default value. This will set all programmed arbitration values for a given requestor to the same value across the blocks that are recommended.

If PRI is set to CSL_BWMNGMT_PRI_NULL (-1) then no change will be made for the corresponding requestors priority level.

If MAXWAIT is set to CSL_BWMNGMT_MAXWAIT_NULL (-1) then no change will be made for the corresponding requestors maxwait setting.

Examples:

```
/* Example 1: sets Priorities and Maxwaits to default values */

CSL_BwmngmtHandle      hBwmngmt;
CSL_BwmngmtHwSetup     hwSetup = CSL_BWMNGMT_HWSETUP_DEFAULTS;

...

// Init successfully done
...
// Open successfully done
...
CSL_bwmngmtHwSetup(hBwmngmt, &hwSetup);
...

/* Example 2: Sets CPU Priority to 1, CPU Maxwait to 8, MAP
 *    Priority to 6 for all blocks (L1D, L2 and EXT)
 */

CSL_BwmngmtHandle      hBwmngmt;
CSL_BwmngmtHwSetup     hwSetup;
hwSetup.cpuPriority = CSL_BWMNGMT_PRI_1;
hwSetup.cpuMaxwait  = CSL_BWMNGMT_MAXWAIT_8;
hwSetup.idmaMaxwait = CSL_BWMNGMT_MAXWAIT_NULL;
hwSetup.slapMaxwait = CSL_BWMNGMT_MAXWAIT_NULL;
hwSetup.mapPriority = CSL_BWMNGMT_PRI_6;
hwSetup.ucMaxwait   = CSL_BWMNGMT_MAXWAIT_NULL;
hwSetup.control     = CSL_BWMNGMT_BLOCK_ALL;
...

// Init successfully done
...
// Open successfully done
...
CSL_bwmngmtHwSetup(hBwmngmt, &hwSetup);
...
```

18.2.5 CSL_bwmngmtGetHwSetup

CSL_Status CSL_bwmngmtGetHwSetup **(** CSL_BwmngmtHandle ***hBwmngmt*****,**
CSL_BwmngmtHwSetup ***** ***hwSetup***
)

Description
Gets the current set up of BWMNGMT.

Arguments

hBwmngmt	Handle to the BWMNGMT instance
hwSetup	Setup structure for BWMNGMT

Return Value
`CSL_Status`

- `CSL_SOK` - Get Hwsetup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - If setup is NULL

Pre Condition
Both CSL_bwmngmtInit() and CSL_bwmngmtOpen() must be called successfully in order before calling this function.

Post Condition
The hardware setup structure is populated with the hardware setup parameters.

Modifies
1. CPU Arbitration Parameters
   - PRI field set in Control Block Specified by "control"
   - MAXWAIT field set in Control Block Specified by "control"

2. IDMA Arbitration Parameter
   - MAXWAIT field set in Control Block Specified by "control"

3. SLAP Arbitration Parameter
   - MAXWAIT field set in Control Block Specified by "control"

4. MAP Arbitration Parameter
   - PRI field set in Control Block Specified by "control" if not EXT then returns CSL_BWMNGMT_PRI_NULL

5. UC Arbitration Parameter
   MAXWAIT field set in Control Block Specified by "control" if not L1D or L2 then returns CSL_BWMNGMT_MAXWAIT_NULL

Example:

```
CSL_BwmngmtHandle     hBwmngmt;
CSL_BwmngmtHwSetup    hwSetup;
CSL_Status            status;

hwSetup.control = CSL_BWMNGMT_BLOCK_L1D;
// Only CSL_BWMNGMT_BLOCK_L1D, CSL_BWMNGMT_BLOCK_L2, or
// CSL_BWMNGMT_BLOCK_EXT are valid
...

// Init successfully done
...
// Open successfully done
...
status = CSL_bwmngmtGetHwSetup(hBwmngmt, &hwSetup);
...
```

18.2.6 CSL_bwmngmtHwControl

CSL_Status CSL_bwmngmtHwControl	**(**	CSL_BwmngmtHandle	***hBwmngmt*,**
		CSL_BwmngmtHwControlCmd	***cmd*,**
		void *	***cmdArg***
	)		

Description
Takes a command of BWMNGMT with an optional argument & implements it. *Not Implemented*. *For future use.*

Arguments

`hBwmngmt`	`Handle to the BWMNGMT instance`
`cmd`	`The command to this API indicates the action to be taken on BWMNGMT.`
`cmdArg`	`An optional argument`

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns.

Pre Condition
Both CSL_bwmngmtInit() and CSL_bwmngmtOpen() must be called successfully in that order before this function can be called

Post Condition
BWMNGMT registers are configured according to the command and the command arguments. The command determines which registers are modified

Modifies
The hardware registers of BWMNGMT

Example

```
CSL_BwmngmtHandle          hBwmngmt;
CSL_BwmngmtHwControlCmd    cmd;
Uint32                     arg;
CSL_Status                 status;

...
status = CSL_bwmngmtHwControl(hBwmngmt, cmd, &arg);
...
```

18.2.7 CSL_bwmngmtGetHwStatus

CSL_Status CSL_bwmngmtGetHwStatus ([CSL_BwmngmtHandle](#) *hBwmngmt*, [CSL_BwmngmtHwStatusQuery](#) *myQuery*, void * *response*)

Description
Gets the status of the different operations of BWMNGMT. *Not Implemented. For future use.*

Arguments

`hBwmngmt`	`Handle to the BWMNGMT instance`
`myQuery`	`The query to this API of BWMNGMT which indicates the status to be returned.`
`response`	`Placeholder to return the status.`

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns.

Pre Condition
Both CSL_bwmngmtInit() and CSL_bwmngmtOpen() must be called successfully in order before calling this function.

Post Condition
None

Modifies
The input argument "response" is modified

Example

```
CSL_BwmngmtHandle          hBwmngmt;
CSL_BwmngmtHwStatusQuery   query;
CSL_Status                 status;
Uint32                     response;
...
status = CSL_bwmngmtGetHwStatus(hBwmngmt, query, &response);
...
```

18.2.8 CSL_bwmngmtGetBaseAddress

CSL_Status CSL_bwmngmtGetBaseAddress(CSL_InstNum bwmngmt*Num*, [CSL_BwmngmtParam](#) * *p*Bwmngmt*Param*, [CSL_BwmngmtBaseAddress](#) * *pBaseAddress*

)

Description

Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_bwmngmtOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

```
bwmngmtNum      Specifies the instance of the BWMNGMT for which
                the base address is requested

pBwmngmtParam   Module specific parameters.

pBaseAddress    Pointer to the base address structure to return
                the base address details.
```

Return Value

`CSL_Status`

- `CSL_SOK` - Successful on getting the base address of BWMNGMT
- `CSL_ESYS_FAIL` - The BWMNGMT instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid parameter.

Pre Condition

None

Post Condition

Base address structure is populated.

Modifies

1. The status variable
2. Base address structure

Example

```
CSL_Status              status;
CSL_BwmngmtBaseAddress  baseAddress;

...
status = CSL_bwmngmtGetBaseAddress(CSL_BWMNGMT,
                                   NULL,
                                   &baseAddress);
...
```

18.3 Data Structures

This section lists the data structures available in the BWMNGMT module.

18.3.1 CSL_BwmngmtObj

Detailed Description
This object contains the reference to the instance of BWMNGMT opened using the CSL_bwmngmtOpen(). The pointer to this object is passed as BWMNGMT handle to all BWMNGMT CSL APIs. CSL_bwmngmtOpen() function initializes this structure based on the parameters passed.

Field Documentation

CSL_InstNum CSL_BwmngmtObj::bwmngmtNum
This is the instance of BWMNGMT being referred to by this object

CSL_BwmngmtRegsOvly CSL_BwmngmtObj::regs
This is a pointer to the registers of the instance of BWMNGMT referred to by CSL_BwmngmtObj object.

18.3.2 CSL_BwmngmtHwSetup

Detailed Description
CSL_BwmngmtHwSetup has all the fields required to configure BWMNGMT.
This structure has the substructures required to configure BWMNGMT at Power-Up/Reset.

Field Documentation

CSL_BwmngmtControlBlocks CSL_BwmngmtHwSetup::control
Controller(s) to be set with Requestors Settings L1D, L2 and/or EXT

CSL_BwmngmtMaxwait CSL_BwmngmtHwSetup::cpuMaxwait
CPU - Requestor Arbitration Settings - MAXWAIT

CSL_BwmngmtPriority CSL_BwmngmtHwSetup::cpuPriority
CPU - Requestor Arbitration Settings - PRI

CSL_BwmngmtMaxwait CSL_BwmngmtHwSetup::idmaMaxwait
IDMA (Internal DMA) Requestor Arbitration Settings - MAXWAIT

CSL_BwmngmtPriority CSL_BwmngmtHwSetup::mapPriority
MAP (Master Port) Requestor Arbitration Settings - PRI

CSL_BwmngmtMaxwait CSL_BwmngmtHwSetup::slapMaxwait
SLAP (Slave Port) Requestor Arbitration Settings - MAXWAIT

CSL_BwmngmtMaxwait CSL_BwmngmtHwSetup::ucMaxwait
UC (User Coherence) Requestor Arbitration Settings – MAXWAIT

18.3.3 CSL_BwmngmtContext

Detailed Description
Bwmngmt specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_BwmngmtContext::contextInfo

Context information of Bwmngmt CSL passed as an argument to CSL_bwmngmtInit(). Present implementation of Bwmngmt CSL doesn't have any context information; hence assigned NULL. The declaration is just a placeholder for future implementation.

18.3.4 CSL_BwmngmtParam

Detailed Description
This is module specific parameter. Present implementation of Bwmngmt CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_BwmngmtParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation. Passed as an argument to CSL_bwmngmtOpen().

18.3.5 CSL_BwmngmtBaseAddress

Detailed Description
This structure contains the base address information for the Bwmngmt instance.

Field Documentation

CSL_BwmngmtRegsOvly CSL_BwmngmtBaseAddress::regs
Base address of the configuration registers of the peripheral

18.4 Enumerations

This section lists the enumerations available in the BWMNGMT module.

18.4.1 CSL_BwmngmtControlBlocks

enum CSL_BwmngmtControlBlocks
Control Block Set for BWMNGMT.
This is used to indicate which control blocks (L1D, L2, and/or EXT) are to be set within BWMNGMT for the given requestor (CPU, IDMA, SLAP, MAP, UC) arbitration settings.

Enumeration values:

CSL_BWMNGMT_BLOCK_ALL	All controller blocks will be update with given requestors arbitration setting
CSL_BWMNGMT_BLOCK_L1D	L1D controller block will be update with given requestors arbitration setting
CSL_BWMNGMT_BLOCK_L2	L2 controller block will be update with given requestors arbitration setting
CSL_BWMNGMT_BLOCK_EXT	EXT controller block will be update with given requestors arbitration setting

18.4.2 CSL_BwmngmtPriority

enum CSL_BwmngmtPriority
Priority Settings for BWMNGMT.
This is used to indicate to set the Priority arbitration settings for the Requestors (CPU, IDMA, SLAP, MAP, UC)

Enumeration values:

CSL_BWMNGMT_PRI_0	Priority arbitration setting 0 - Highest priority requestor
CSL_BWMNGMT_PRI_1	Priority arbitration setting 1 - 2nd Highest priority requestor
CSL_BWMNGMT_PRI_2	Priority arbitration setting 2 - 3rd Highest priority requestor
CSL_BWMNGMT_PRI_3	Priority arbitration setting 3 - 4th Highest priority requestor
CSL_BWMNGMT_PRI_4	Priority arbitration setting 4 - 5th Highest priority requestor
CSL_BWMNGMT_PRI_5	Priority arbitration setting 5 - 6th Highest priority requestor
CSL_BWMNGMT_PRI_6	Priority arbitration setting 6 - 7th Highest priority requestor
CSL_BWMNGMT_PRI_7	Priority arbitration setting 7 - Lowest priority requestor
CSL_BWMNGMT_PRI_NULL	Priority arbitration setting NULL - Due Not Program PRIORITY for this requestor

18.4.3 CSL_BwmngmtMaxwait

enum CSL_BwmngmtMaxwait
Maxwait Settings for BWMNGMT.
This is used to indicate to set Maxwait arbitration settings for the Requestors (CPU, IDMA, SLAP, MAP, UC).

Enumeration values:

CSL_BWMNGMT_MAXWAIT_0	Maxwait arbitration setting 0 - Always stall due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_1	Maxwait arbitration setting 1 - Stall max of 1 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_2	Maxwait arbitration setting 2 - Stall max of 2 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_4	Maxwait arbitration setting 4 - Stall max of 4 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_8	Maxwait arbitration setting 8 - Stall max of 8 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_16	Maxwait arbitration setting 16 - Stall max of 16 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_32	Maxwait arbitration setting 32 - Stall max of 32 cycle due to higher priority requestor
CSL_BWMNGMT_MAXWAIT_NULL	Maxwait arbitration setting NULL - Due Not Program MAXWAIT for this requestor

18.4.4 CSL_BwmngmtHwStatusQuery

enum CSL_BwmngmtHwStatusQuery
Enumeration for Hardware status query

Enumeration values:

PLACEHOLDER0	Placeholder for future implementation

18.4.5 CSL_BwmngmtHwControlCmd

enum CSL_BwmngmtHwControlCmd
Enumeration for Hardware control command

Enumeration values:

PLACEHOLDER2	Placeholder for future implementation

18.5 Macros

**#define CSL_BWMNGMT_HWSETUP_DEFAULTS **

```
{  \
        (CSL_BwmngmtPriority)CSL_BWMNGMT_CPUARBL1D_PRI_RESETVAL,      \
        (CSL_BwmngmtMaxwait)CSL_BWMNGMT_CPUARBL1D_MAXWAIT_RESETVAL,  \
        (CSL_BwmngmtMaxwait)CSL_BWMNGMT_IDMAARBL2_MAXWAIT_RESETVAL,  \
        (CSL_BwmngmtMaxwait)CSL_BWMNGMT_SLAPARBL1D_MAXWAIT_RESETVAL, \
        (CSL_BwmngmtPriority)CSL_BWMNGMT_MAPARBEXT_PRI_RESETVAL,      \
        (CSL_BwmngmtMaxwait)CSL_BWMNGMT_UCARBL1D_MAXWAIT_RESETVAL,   \
        (CSL_BwmngmtControlBlocks)CSL_BWMNGMT_BLOCK_ALL               \
        }
```

The default values for the hardware setup of bwmngmt

18.6 Typedefs

typedef CSL_BwmngmtObj * CSL_BwmngmtHandle

This is a pointer to CSL_BwmngmtObj & is passed as the first parameter to all BWMNGMT CSL APIs

Chapter 19
CACHE Module

Topics

19.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within CACHE module.

This module use three cache architectures, Level 1 Program (L1P), Level 1 Data (L1D) and Level 2 CACHE architectures. The L1P and L1D can be configured as 0K, 4K, 8K, 16K, and 32K CACHE size. The L2 can be configured as 32KB, 64KB, 128KB, and 256KB CACHE size. This CACHE module supports the Block and Global Coherence Operations.

19.2 Functions

This section lists the functions available in the CACHE module.

19.2.1 CACHE_enableCaching

void CACHE_enableCaching (CE_MAR *mar*)

Description
This function enables caching for the specified block of memory. This is accomplished by setting the PC bit in the appropriate Memory Attribute Register (MAR). By default, caching is disabled for all memory spaces.

Arguments

`mar`	`EMIF range,` specifies a block of external memory to enable caching

Return Value
None

Pre Condition
None

Post Condition
Caching for the specified memory range is enabled.

Modifies
MAR registers

Example

```
...
CACHE_enableCaching(CACHE_EMIFB_CE00);
...
```

19.2.2 CACHE_wait

void CACHE_wait (void)

Description
This function waits for the previously issued block operations to complete. This does a partial wait i.e. waits for the cache status register to read back as done.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
...
CACHE_wait();
...
```

19.2.3 CACHE_waitInternal

void CACHE_waitInternal (void)

Description
This function waits for previously issued block operations to complete. This does a partial wait i.e. waits for the cache status register to read back as done.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
...

CACHE_waitInternal();
...
```

19.2.4 CACHE_freezeL1

CACHE_L1_Freeze CACHE_freezeL1 (void)

Description
This function freezes the L1P and L1D Cache

As per the specification,
1. The new freeze state is programmed in L1DCC, L1PCC
2. The old state is read from the L1DCC, L1PCC from the POPER field.

This latter read accomplishes two things viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1_FREEZE` - Old Freeze State of L1 Cache
- `CACHE_L1P_FREEZE` - Old Freeze State of L1P Cache
- `CACHE_L1D_FREEZE` - Old Freeze State of L1D Cache
- `CACHE_L1_NORMAL` - Normal State of L1 Cache

Pre Condition
The CACHE_enableCaching(), CACHE_setL1pSize() and CACHE_setL1dSize() must be called successfully in that order before calling this API

Post Condition

Freeze L1 cache

Modifies
L1DCC and L1PCC registers

Example
```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_freezeL1();
...
```

19.2.5 CACHE_unfreezeL1

CACHE_L1_Freeze CACHE_unfreezeL1 (void)

Description
This function unfreezes the L1P and L1D Cache.

As per the specification,
1. The new unfreeze state is programmed in L1DCC, L1PCC.
2. The old state is read from the L1DCC, L1PCC from the POPER field.

This latter read accomplishes 2 things viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1_FREEZE` - Old Freeze State of L1 Cache
- `CACHE_L1P_FREEZE` - Old Freeze State of L1P Cache
- `CACHE_L1D_FREEZE` - Old Freeze State of L1D Cache
- `CACHE_L1_NORMAL` - Normal State of L1 Cache

Pre Condition
The CACHE_enableCaching(), CACHE_setL1pSize() and CACHE_setL1dSize() must be called successfully in that order before calling this API

Post Condition
Unfreeze the L1 cache

Modifies
L1DCC and L1PCC registers

Example
```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_unfreezeL1();
...
```

19.2.6 CACHE_setL1pSize

CACHE_L1Size CACHE_setL1pSize (CACHE_L1Size *newSize*)

Description
This function sets the L1P cache size. The configurable L1P cache sizes are 0K, 4K, 8K, 16K, and 32K.

As per the specification,
1. The new size is programmed in L1PCFG.
2. L1PCFG is read back to ensure it is set.

Arguments

`newSize` `New Cache size to be programmed`

Return Value CACHE_L1Size

- Old size of L1 Cache

Pre Condition
The CACHE must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition

Set L1P cache size

Modifies
L1PCFG register

Example
```
...
CACHE_L1Size oldSize;
oldSize = CACHE_setL1pSize(CACHE_L1_32KCACHE);
...
```

19.2.7 CACHE_freezeL1p

CACHE_L1_Freeze CACHE_freezeL1p (void)

Description
This function freezes L1P Cache.
As per the specification,

1. The new freeze state is programmed in L1PCC.
2. The old state is read from the L1PCC from the POPER field.

This latter read accomplishes two things; viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1P_FREEZE` - Old Freeze State of L1P Cache
- `CACHE_L1P_NORMAL` - Normal State of L1P Cache

Pre Condition
The CACHE_enableCaching() and CACHE_setL1pSize() must be called successfully in that order before calling this API

Post Condition

Freeze L1P cache

Modifies
L1PCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_freezeL1p();
...
```

19.2.8 CACHE_unfreezeL1p

CACHE_L1_Freeze CACHE_unfreezeL1p (void)

Description
This function unfreezes L1P Cache.

As per the specification,
1. The normal state is programmed in L1PCC
2. The old state is read from the L1PCC from the POPER field.

This latter read accomplishes two things, viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1P_FREEZE` - Old Freeze State of L1P Cache
- `CACHE_L1P_NORMAL` - Normal State of L1P Cache

Pre Condition
The CACHE_enableCaching() and CACHE_setL1pSize() must be called successfully in that order before calling this API.

Post Condition

Unreeze L1P cache

Modifies
L1PCC register

Example
```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_unfreezeL1p();
...
```

19.2.9 CACHE_invL1p

void CACHE_invL1p	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L1P block invalidate command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

As per the specification,

1. The start of the range that needs to be invalidated is written into L1PIBAR
2. The byte count is programmed in L1PIWC.

Arguments
```
blockPtr    Start address of range to be invalidated
```

```
byteCnt     Number of bytes to be invalidated

wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL1pSize()* must be called successfully in that order before calling this API.

Post Condition

Invalidate L1P cache

Modifies
L1PIBAR and L1PIWC registers

Example

```
...
CACHE_L1Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL1pSize(CACHE_L1_32KCACHE);
CACHE_invL1p((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
...
```

19.2.10 CACHE_invAllL1p

void CACHE_invAllL1p (CACHE_Wait ***wait***)

Description
This function issues an L1P invalidate all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,
1. The L1PINV is programmed

Arguments

```
wait        Wait flag

            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL1pSize()* must be called successfully in that order before calling this API

Post Condition

Invalidate all L1P cache

Modifies
L1PINV register

Example
```
...
CACHE_invAllL1p(CACHE_NOWAIT);
...
```

19.2.11 CACHE_setL1dSize

CACHE_L1Size CACHE_setL1dSize (CACHE_L1Size *newSize*)

Description
This function sets the size of the L1D cache. The configurable L1D cache sizes are 0K, 4K, 8K, 16K, and 32K.

As per the specification,
1. The new size is programmed in L1DCFG
2. L1DCFG is read back to ensure it is set.

Arguments

```
newSize      New size to be programmed
```

Return Value
`CACHE_L1Size`

- Old L1D Cache Size

Pre Condition
The CACHE must be successfully enabled via *CACHE_enableCaching()* before calling this function

Post Condition
Set L1D cache size

Modifies
L1DCFG register

Example

```
...
CACHE_L1Size oldSize;
oldSize = CACHE_setL1dSize(CACHE_L1_32KCACHE);
...
```

19.2.12 CACHE_freezeL1d

CACHE_L1_Freeze CACHE_freezeL1d (void)

Description
This function freezes L1D Cache.

As per the specification,
1.The new freeze state is programmed in L1DCC.
2. The old state is read from the L1DCC from the POPER field.
This latter read accomplishes two things; viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1D_FREEZE` - Old Freeze State of L1D Cache
- `CACHE_L1D_NORMAL` - Normal State of L1D Cache

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition
Freeze L1D cache

Modifies
L1DCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_freezeL1d();
...
```

19.2.13 CACHE_unfreezeL1d

CACHE_L1_Freeze CACHE_unfreezeL1d (void)

Description
This API Unfreezes L1D Cache. As per the specification,
1. The normal state is programmed in L1DCC
2. The old state is read from the L1DCC from the POPER field.

This latter read accomplishes two things; viz. ensuring the new state is programmed as well as reading the old programmed value.

Arguments
None

Return Value
`CACHE_L1_Freeze`

- `CACHE_L1D_FREEZE` - Old Freeze State of L1D Cache
- `CACHE_L1D_NORMAL` - Normal State of L1D Cache

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API.

Post Condition

Unfreeze L1D cache

Modifies
L1DCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;
oldFreezeState = CACHE_unfreezeL1d();
...
```

19.2.14 CACHE_wbL1d

void CACHE_wbL1d	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L1D block writeback command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

As per the specification,

1. The start of the range that needs to be written back is programmed into L1DWBAR.
2. The byte count is programmed in L1DWWC.

Arguments

```
blockPtr    Start address of range to be written back

byteCnt     Number of bytes to be written back

wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition

Writeback L1D cache

Modifies
L1DWWC and L1DWBAR registers

Example

```
...
CACHE_L1Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL1dSize(CACHE_L1_32KCACHE);
CACHE_wbL1d((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
...
```

19.2.15 CACHE_invL1d

void CACHE_invL1d	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L1D block invalidate command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

As per the specification,

1. The start of the range that needs to be invalidated is written into L1DIBAR.
2. The byte count is programmed in L1DIWC.

Arguments

```
blockPtr    Start address of range to be invalidated

byteCnt     Number of bytes to be invalidated

wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition

Invalidates the L1D cache

Modifies
L1DIWC and L1DIBAR registers

Example

```
...
CACHE_L1Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL1dSize(CACHE_L1_32KCACHE);
CACHE_invL1d ((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
```

19.2.16 CACHE_wbInvL1d

void CACHE_wbInvL1d	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L1D block writeback and invalidate command to the cache controller. If Writeback invalidates range specified in L1D. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only.

As per the specification,

1. The start of the range that needs to be writeback invalidated is programmed into L1DWIBAR.
2. The byte count is programmed in L1DWIWC.

Arguments

```
blockPtr    Start address of range to be written back
            invalidated

byteCnt     Number of bytes to be written back invalidated

wait        Wait flag

            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API.

Post Condition
Writeback and invalidates the L1D cache

Modifies
L1DWIWC and L1DWIBAR registers

Example

```
...
CACHE_L1Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL1dSize(CACHE_L1_32KCACHE);
CACHE_wbInvL1d ((Uint32*)(0xC0000000),200,CACHE_NOWAIT);
...
```

19.2.17 CACHE_wbAllL1d

void CACHE_wbAllL1d (CACHE_Wait *wait*)

Description
This function issues an L1D writeback all command to the cache controller. If Writeback All of L1D of a previous cache operation is still active, then the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,

1. The L1DWB is programmed.

Arguments

```
wait          Wait flag

              CACHE_NOWAIT – return immediately
              CACHE_WAIT – wait until the operation completes
              CACHE_WAITINTERNAL – wait until the relevant cache
              status registers indicate completion
```

Return Value
None

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition
Writeback all the L1D cache

Modifies
L1DWB register

Example

```
...
CACHE_wbAllL1d(CACHE_NOWAIT);
...
```

19.2.18 CACHE_invAllL1d

void CACHE_invAllL1d ([CACHE_Wait](#) *wait*)

Description
This function issues an L1D invalidate all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,

1. The L1DINV is programmed.

Arguments

```
wait          Wait flag

              CACHE_NOWAIT – return immediately
              CACHE_WAIT – wait until the operation completes
              CACHE_WAITINTERNAL – wait until the relevant cache
              status registers indicate completion
```

Return Value
None

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition
Invalidates the all L1D cache

Modifies
L1DINV register

Example

```
...
CACHE_invAllL1d(CACHE_NOWAIT);
...
```

19.2.19 CACHE_wbInvAllL1d

void CACHE_wbInvAllL1d (CACHE_Wait *wait*)

Description
This function issues an L1D writeback and invalidate all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,
1. The L1DWBINV is programmed.

Arguments

```
wait          Wait flag

              CACHE_NOWAIT – return immediately
              CACHE_WAIT – wait until the operation completes
              CACHE_WAITINTERNAL – wait until the relevant cache
              status registers indicate completion
```

Return Value
None

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL1dSize()* must be called successfully in that order before calling this API

Post Condition

Writeback and invalidates all L1D cache

Modifies
L1DWBINV register

Example

```
...
CACHE_wbInvAllL1d(CACHE_NOWAIT);
...
```

19.2.20 CACHE_setL2Size

CACHE_L2Size CACHE_setL2Size (CACHE_L2Size *newSize*)

Description
This function sets the L2 Cache size. The configurable L2 cache sizes are 32KB, 64KB, 128KB, and 256KB.

As per the specification,
1. The old size is read from the L2CFG.
2. The new size is programmed in L2CFG.
3. L2CFG is read back to ensure it is set.

Arguments

```
newSize      New memory size to be programmed
```

Return Value
`CACHE_L2Size`

- Old L2 Cache Size

Pre Condition
The CACHE must be successfully enabled via *CACHE_enableCaching()* before calling this function

Post Condition

Sets the L2 cache size

Modifies
L2CFG register

Example
```
...
CACHE_L2Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
...
```

19.2.21 CACHE_setL2Mode

CACHE_L2Mode CACHE_setL2Mode (CACHE_L2Mode *newMode*)

Description
This function sets the L2 Cache mode. The configurable L2 Cache modes are Normal and Freeze mode.

As per the specification,
1. The old mode is read from the L2CFG.
2. The new mode is programmed in L2CFG.
3. L2CFG is read back to ensure it is set.

Arguments

```
newMode      New mode to be programmed
```

Return Value
`CACHE_L2Mode`

- Old Mode set for L2

Pre Condition
The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Set L2 cache mode

Modifies
L2CFG register

Example

```
...
CACHE_L2Mode oldMode;
CACHE_L2Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
oldMode = CACHE_setL2Mode(CACHE_L2_NORMAL);
...
```

19.2.22 CACHE_wbL2

void CACHE_wbL2	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L2 block writeback command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

As per the specification,

1. The start of the range that needs to be written back is programmed into L2WBAR.
2. The byte count is programmed in L2WWC

Arguments

```
blockPtr    Start address of range to be written back
```

```
byteCnt      Number of bytes to be written back

wait         Wait flag
             CACHE_NOWAIT – return immediately
             CACHE_WAIT – wait until the operation completes
             CACHE_WAITINTERNAL – wait until the relevant cache
             status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Writeback the L2 cache

Modifies
L2WWC and L2WBAR registers

Example

```
...
CACHE_L2Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
CACHE_wbL2((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
...
```

19.2.23 CACHE_invL2

void CACHE_invL2	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L2 block invalidate command to the cache controller. If a previous cache peration is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

As per the specification,

1. The start of the range that needs to be written back is programmed into L2IBAR
2. The byte count is programmed in L2IWC.

Arguments

```
blockPtr    Start address of range to be invalidated

byteCnt     Number of bytes to be invalidated

wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Invalidates the L2 cache

Modifies
L2IBAR and L2IWC registers

Example

```
...
CACHE_L2Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
CACHE_invL2((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
...
```

19.2.24 CACHE_wbInvL2

void CACHE_wbInvL2	**(**	**void ***	***blockPtr*,**
		Uint32	***byteCnt*,**
		CACHE_Wait	***wait***
	)		

Description
This function issues an L2 block writeback and invalidate command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed. Although the block size can be specified in number of bytes, the cache controller operates on whole cache lines only. To prevent unintended behavior, blockPtr and byteCnt should be multiples of the cache line size.

As per the specification,

1. The start of the range that needs to be written back is programmed into L2WIBAR
2. The byte count is programmed in L2WIWC.

Arguments

```
blockPtr    Start address of range to be written back invalidated

byteCnt     Number of bytes to be written back invalidated

wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Writeback and invalidates the L2 cache

Modifies
L2WIBAR and L2WIWC registers

Example

```
...
CACHE_L2Size oldSize;
CACHE_enableCaching(CACHE_EMIFA_CE40);
oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
CACHE_wbInvL2((Uint32*)(0xC0000000), 200, CACHE_NOWAIT);
...
```

19.2.25 CACHE_wbAllL2

void CACHE_wbAllL2 (CACHE_Wait *wait*)

Description
This function issues an L2 writeback all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,

1. The L2WB needs to be programmed.

Arguments

```
wait        Wait flag

            CACHE_NOWAIT – return immediately
```

```
CACHE_WAIT – wait until the operation completes
CACHE_WAITINTERNAL – wait until the relevant cache
status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Writeback all L2 cache

Modifies
L2WB register

Example

```
...
CACHE_wbAllL2(CACHE_NOWAIT);
...
```

19.2.26 CACHE_invAllL2

void CACHE_invAllL2 (CACHE_Wait *wait*)

Description
This function issues an L2 invalidate all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,
1. The L2INV needs to be programmed.

Arguments

```
wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition
Invalidates all L2 cache

Modifies
L2INV register

Example

```
...
CACHE_invAllL2(CACHE_NOWAIT);
...
```

19.2.27 CACHE_wbInvAllL2

void CACHE_wbInvAllL2 (CACHE_Wait *wait*)

Description
This function issues an L2 writeback and invalidate all command to the cache controller. If a previous cache operation is still active, the function waits for its completion before initiating the new operation, in order to prevent lockout of interrupts. If the argument wait is CACHE_NOWAIT, then the function returns immediately, regardless of whether the operation has completed.

As per the specification,

1. The L2WBINV needs to be programmed.

Arguments

```
wait        Wait flag
            CACHE_NOWAIT – return immediately
            CACHE_WAIT – wait until the operation completes
            CACHE_WAITINTERNAL – wait until the relevant cache
            status registers indicate completion
```

Return Value
None

Pre Condition

The *CACHE_enableCaching()* and *CACHE_setL2Size()* must be called successfully in that order before calling this API

Post Condition

Writeback and invalidates all the L2 cache

Modifies
L2WBINV register

Example

```
...
CACHE_wbInvAllL2(CACHE_NOWAIT);
...
```

19.3 Enumerations

This section lists the enumerations available in the CACHE module.

19.3.1 CE_MAR

enum CE_MAR
Enumeration for Emif ranges. This is used for setting up the cache ability of the EMIF ranges.

Enumeration values:

CACHE_EMIFA_CE20	EMIF ranges from 0xA0000000 – 0xA0FFFFFF
CACHE_EMIFA_CE21	EMIF ranges from 0xA1000000 – 0xA1FFFFFF
CACHE_EMIFA_CE22	EMIF ranges from 0xA2000000 – 0xA2FFFFFF
CACHE_EMIFA_CE23	EMIF ranges from 0xA3000000 – 0xA3FFFFFF
CACHE_EMIFA_CE24	EMIF ranges from 0xA4000000 – 0xA4FFFFFF
CACHE_EMIFA_CE25	EMIF ranges from 0xA5000000 – 0xA5FFFFFF
CACHE_EMIFA_CE26	EMIF ranges from 0xA6000000 – 0xA6FFFFFF
CACHE_EMIFA_CE27	EMIF ranges from 0xA7000000 – 0xA7FFFFFF
CACHE_EMIFA_CE28	EMIF ranges from 0xA8000000 – 0xA8FFFFFF
CACHE_EMIFA_CE29	EMIF ranges from 0xA9000000 – 0xA9FFFFFF
CACHE_EMIFA_CE210	EMIF ranges from 0xAA000000 – 0xAAFFFFFF
CACHE_EMIFA_CE211	EMIF ranges from 0xAB000000 – 0xABFFFFFF
CACHE_EMIFA_CE212	EMIF ranges from 0xAC000000 – 0xACFFFFFF
CACHE_EMIFA_CE213	EMIF ranges from 0xAD000000 – 0xADFFFFFF
CACHE_EMIFA_CE214	EMIF ranges from 0xAE000000 – 0xAEFFFFFF
CACHE_EMIFA_CE215	EMIF ranges from 0xAF000000 – 0xAFFFFFFF
CACHE_EMIFA_CE30	EMIF ranges from 0xB0000000 – 0xB0FFFFFF
CACHE_EMIFA_CE31	EMIF ranges from 0xB1000000 – 0xB1FFFFFF
CACHE_EMIFA_CE32	EMIF ranges from 0xB2000000 – 0xB2FFFFFF
CACHE_EMIFA_CE33	EMIF ranges from 0xB3000000 – 0xB3FFFFFF
CACHE_EMIFA_CE34	EMIF ranges from 0xB4000000 – 0xB4FFFFFF
CACHE_EMIFA_CE35	EMIF ranges from 0xB5000000 – 0xB5FFFFFF
CACHE_EMIFA_CE36	EMIF ranges from 0xB6000000 – 0xB6FFFFFF
CACHE_EMIFA_CE37	EMIF ranges from 0xB7000000 – 0xB7FFFFFF
CACHE_EMIFA_CE38	EMIF ranges from 0xB8000000 – 0xB8FFFFFF
CACHE_EMIFA_CE39	EMIF ranges from 0xB9000000 – 0xB9FFFFFF
CACHE_EMIFA_CE310	EMIF ranges from 0xBA000000 – 0xBAFFFFFF
CACHE_EMIFA_CE311	EMIF ranges from 0xBB000000 – 0xBBFFFFFF
CACHE_EMIFA_CE312	EMIF ranges from 0xBC000000 – 0xBCFFFFFF
CACHE_EMIFA_CE313	EMIF ranges from 0xBD000000 – 0xBDFFFFFF
CACHE_EMIFA_CE314	EMIF ranges from 0xBE000000 – 0xBEFFFFFF
CACHE_EMIFA_CE315	EMIF ranges from 0xBF000000 – 0xBFFFFFFF
CACHE_EMIFA_CE40	EMIF ranges from 0xC0000000 – 0xC0FFFFFF
CACHE_EMIFA_CE41	EMIF ranges from 0xC1000000 – 0xC1FFFFFF
CACHE_EMIFA_CE42	EMIF ranges from 0xC2000000 – 0xC2FFFFFF
CACHE_EMIFA_CE43	EMIF ranges from 0xC3000000 – 0xC3FFFFFF
CACHE_EMIFA_CE44	EMIF ranges from 0xC4000000 – 0xC4FFFFFF
CACHE_EMIFA_CE45	EMIF ranges from 0xC5000000 – 0xC5FFFFFF

CACHE_EMIFA_CE46	EMIF ranges from 0xC6000000 – 0xC6FFFFFF
CACHE_EMIFA_CE47	EMIF ranges from 0xC7000000 – 0xC7FFFFFF
CACHE_EMIFA_CE48	EMIF ranges from 0xC8000000 – 0xC8FFFFFF
CACHE_EMIFA_CE49	EMIF ranges from 0xC9000000 – 0xC9FFFFFF
CACHE_EMIFA_CE410	EMIF ranges from 0xCA000000 – 0xCAFFFFFF
CACHE_EMIFA_CE411	EMIF ranges from 0xCB000000 – 0xCBFFFFFF
CACHE_EMIFA_CE412	EMIF ranges from 0xCC000000 – 0xCCFFFFFF
CACHE_EMIFA_CE413	EMIF ranges from 0xCD000000 – 0xCDFFFFFF
CACHE_EMIFA_CE414	EMIF ranges from 0xCE000000 – 0xCEFFFFFF
CACHE_EMIFA_CE415	EMIF ranges from 0xCF000000 – 0xCFFFFFFF
CACHE_EMIFA_CE50	EMIF ranges from 0xD0000000 – 0xD0FFFFFF
CACHE_EMIFA_CE51	EMIF ranges from 0xD1000000 – 0xD1FFFFFF
CACHE_EMIFA_CE52	EMIF ranges from 0xD2000000 – 0xD2FFFFFF
CACHE_EMIFA_CE53	EMIF ranges from 0xD3000000 – 0xD3FFFFFF
CACHE_EMIFA_CE54	EMIF ranges from 0xD4000000 – 0xD4FFFFFF
CACHE_EMIFA_CE55	EMIF ranges from 0xD5000000 – 0xD5FFFFFF
CACHE_EMIFA_CE56	EMIF ranges from 0xD6000000 – 0xD6FFFFFF
CACHE_EMIFA_CE57	EMIF ranges from 0xD7000000 – 0xD7FFFFFF
CACHE_EMIFA_CE58	EMIF ranges from 0xD8000000 – 0xD8FFFFFF
CACHE_EMIFA_CE59	EMIF ranges from 0xD9000000 – 0xD9FFFFFF
CACHE_EMIFA_CE510	EMIF ranges from 0xDA000000 – 0xDAFFFFFF
CACHE_EMIFA_CE511	EMIF ranges from 0xDB000000 – 0xDBFFFFFF
CACHE_EMIFA_CE512	EMIF ranges from 0xDC000000 – 0xDCFFFFFF
CACHE_EMIFA_CE513	EMIF ranges from 0xDD000000 – 0xDDFFFFFF
CACHE_EMIFA_CE514	EMIF ranges from 0xDE000000 – 0xDEFFFFFF
CACHE_EMIFA_CE515	EMIF ranges from 0xDF000000 – 0xDFFFFFFF
CACHE_EMIFB_CE00	EMIF ranges from 0xE0000000 – 0xE0FFFFFF
CACHE_EMIFB_CE01	EMIF ranges from 0xE1000000 – 0xE1FFFFFF
CACHE_EMIFB_CE02	EMIF ranges from 0xE2000000 – 0xE2FFFFFF
CACHE_EMIFB_CE03	EMIF ranges from 0xE3000000 – 0xE3FFFFFF
CACHE_EMIFB_CE04	EMIF ranges from 0xE4000000 – 0xE4FFFFFF
CACHE_EMIFB_CE05	EMIF ranges from 0xE5000000 – 0xE5FFFFFF
CACHE_EMIFB_CE06	EMIF ranges from 0xE6000000 – 0xE6FFFFFF
CACHE_EMIFB_CE07	EMIF ranges from 0xE7000000 – 0xE7FFFFFF
CACHE_EMIFB_CE08	EMIF ranges from 0xE8000000 – 0xE8FFFFFF
CACHE_EMIFB_CE09	EMIF ranges from 0xE9000000 – 0xE9FFFFFF
CACHE_EMIFB_CE010	EMIF ranges from 0xEA000000 – 0xEAFFFFFF
CACHE_EMIFB_CE011	EMIF ranges from 0xEB000000 – 0xEBFFFFFF
CACHE_EMIFB_CE012	EMIF ranges from 0xEC000000 – 0xECFFFFFF
CACHE_EMIFB_CE013	EMIF ranges from 0xED000000 – 0xEDFFFFFF
CACHE_EMIFB_CE014	EMIF ranges from 0xEE000000 – 0xEEFFFFFF
CACHE_EMIFB_CE015	EMIF ranges from 0xEF000000 – 0xEFFFFFFF

19.3.2 CACHE_Wait

enum CACHE_Wait
Enumeration for Cache wait flags.
This is used for specifying whether the cache operations should block till the desired operation is complete.

Enumeration values:

CACHE_NOWAIT	No blocking, the call exits after programming the control registers.
CACHE_WAITINTERNAL	Blocking Call, the call exits after the relevant cache status registers indicate completion.
CACHE_WAIT	Blocking Call, the call waits not only till the cache status registers indicate completion, but also till a write read is issued to the EMIF registers (if required) .

19.3.3 CACHE_L1_Freeze

enum CACHE_L1_Freeze
Enumeration for Cache Freeze flags. This is used for reporting back the current state of the L1.

Enumeration values:

CACHE_L1D_NORMAL	L1D is in Normal State
CACHE_L1D_FREEZE	L1D is in Freeze State
CACHE_L1P_NORMAL	L1P is in Normal State
CACHE_L1P_FREEZE	L1P is in Freeze State
CACHE_L1_NORMAL	L1D, L1P is in Normal State
CACHE_L1_FREEZE	L1D, L1P is in Freeze State

19.3.4 CACHE_L1Size

enum CACHE_L1Size
Enumeration for L1P or L1D Sizes.

Enumeration values:

CACHE_L1_0KCACHE	No Cache
CACHE_L1_4KCACHE	4KB Cache
CACHE_L1_8KCACHE	8KB Cache
CACHE_L1_16KCACHE	16KB Cache
CACHE_L1_32KCACHE	32KB Cache

19.3.5 CACHE_L2Size

enum CACHE_L2Size
Enumeration for L2 Sizes.

Enumeration values:

CACHE_0KCACHE	No Cache
CACHE_32KCACHE	32KB Cache
CACHE_64KCACHE	64KB Cache
CACHE_128KCACHE	128KB Cache

CACHE_256KCACHE	256KB Cache

19.3.6 CACHE_L2Mode

enum CACHE_L2Mode
Enumeration for L2 Modes.

Enumeration values:

CACHE_L2_NORMAL	Enabled/Normal Mode
CACHE_L2_FREEZE	Freeze Mode

19.4 Macros

#define CACHE_L1D_LINESIZE 64
L1D Line Size

#define CACHE_L1P_LINESIZE 32
L1P Line Size

#define CACHE_L2_LINESIZE 128
L2 Line Size

**#define CACHE_ROUND_TO_LINESIZE (CACHE, ELCNT, ELSIZE) **

```
((CACHE_##CACHE##_LINESIZE *                          \
        ((ELCNT)*(ELSIZE)/CACHE_##CACHE##_LINESIZE) + 1) / (ELSIZE))
```

Cache Round to Line size

Chapter 20
CFG Module

Topics

20.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within CFG module.

This module provides memory protection for Internal configration space. If Invalid write accesses to reserved regions of Internal configuration Space will generate an exception.If a serious of protection faults occurs to the CFG space, only that first such violation is recorded and only one exception is generated via the Extended Memory Controller CPU memory protection fault interrupt.Once this fault is cleared, a new protection violation will result in its information being recorded and new exception being generated.

20.2 Functions

This section lists the functions available in the CFG module.

20.2.1 CSL_cfgInit

CSL_Status CSL_cfgInit (CSL_CfgContext * *pContext*)

Description
This is the initialization function for the CFG CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext   Context information for the instance. Should be NULL
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
CSL_cfgInit(NULL);
...
```

20.2.2 CSL_cfgOpen

CSL_CfgHandle CSL_cfgOpen (CSL_CfgObj * *pCfgObj*, CSL_InstNum *cfgNum*, CSL_CfgParam * *pCfgParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the instance and returns a handle to the instance. The open call sets up the data structures for the particular instance of CFG device. The device can be re-opened anytime after it has been normally closed, if so required. The handle returned by this call is input as an essential argument for rest of the APIs described for this module.

Arguments

`pCfgObj`	`Pointer to the CFG instance object`
`cfgNum`	`Instance of the CFG to be opened.`
`pCfgParam`	`Pointer to module specific parameters`
`pStatus`	`Pointer for returning status of the function call`

Return Value
`CSL_CfgHandle`

- Valid CFG instance handle will be returned if status value is equal to CSL_SOK.

Pre Condition
CSL_cfgInit() has to be called before calling this function.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid CFG handle is returned.
- `CSL_ESYS_FAIL` - The CFG instance is invalid.
- `CSL_ESYS_INVPARAMS` - The Obj structure passed is invalid

2. CFG object structure is populated.

Modifies
1. The status variable
2. CFG object structure

Example

```
CSL_Status          status;
CSL_CfgObj          cfgObj;
CSL_CfgHandle       hCfg;

...
hCfg = CSL_cfgOpen(&cfgObj, CSL_MEMPROT_CONFIG, NULL, &status);
...
```

20.2.3 CSL_cfgClose

CSL_Status CSL_cfgClose (CSL_CfgHandle *hCfg*)

Description
This function closes the specified instance of CFG.

Arguments

`hCfg`	`Handle to the CFG instance`

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_cfgInit() and CSL_cfgOpen() must be called successfully in order before calling CSL_cfgClose().

Post Condition
The CFG CSL APIs can not be called until the CFG CSL is reopened again using CSL_cfgOpen().

Modifies
CSL_cfgObj structure values

Example
```
CSL_CfgHandle     hCfg;
CSL_Status        status;
...
status = CSL_cfgClose(hCfg);
...
```

20.2.4 CSL_cfgHwControl

CSL_Status CSL_cfgHwControl	**(**	**CSL_CfgHandle**	***hCfg*,**
		CSL_CfgHwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
Takes a command of CFG with an optional argument and implements it.

Arguments

`hCfg`	`Handle to the CFG instance`
`cmd`	`The command to this API indicates the action to be taken on CFG.`
`arg`	`An optional argument.`

Return Value
`CSL_Status`

- `CSL_SOK` - Command successful.
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_cfgInit() and CSL_cfgOpen() must be called successfully in order before calling CSL_cfgHwControl*()*.

Post Condition
CFG registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
The registers of CFG.

Example

```
CSL_CfgHandle          hCfg;
CSL_Status             status;
...
status = CSL_cfgHwControl(hCfg, CSL_CFG_CMD_CLEAR, NULL);
```

20.2.5 CSL_cfgGetHwStatus

CSL_Status CSL_cfgGetHwStatus **(** [CSL_CfgHandle](#) ***hCfg*****,**
[CSL_CfgHwStatusQuery](#) ***query*****,**
void * ***response***
)

Description
Gets the status of the different operations of CFG.

Arguments

`hCfg`	`Handle to the CFG instance`
`query`	`The query to this API of CFG which indicates the status to be returned.`
`response`	`Placeholder to return the status.`

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful
- `CSL_ESYS_INVQUERY` - Invalid query command
- `CSL_ESYS_INVPARAMS` - Invalid parameter
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_cfgInit() and CSL_cfgOpen() must be called successfully in order before calling CSL_cfgGetHwStatus().

Post Condition
None

Modifies
Third parameter “response” vlaue

Example

```
CSL_CfgHandle          hCfg;
Uint32                 response;
CSL_Status             status;
...
status = CSL_cfgGetHwStatus(hCfg,
                                  CSL_CFG_QUERY_FAULT_ADDR,
                                  &response);
...
```

20.2.6 CSL_cfgGetBaseAddress

CSL_Status CSL_cfgGetBaseAddress (CSL_InstNum **cfg*Num*,**
CSL_CfgParam * ***pCfgParam*,**
CSL_CfgBaseAddress * ***pBaseAddress***
)

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_cfgOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral MMRs go to an alternate location.

Arguments

`cfgNum`	`Specifies the instance of the CFG for which the base address is requested`
`pCfgParam`	`Module specific parameters`
`pBaseAddress`	`Pointer to the base address structure to return the base address details`

Return Value
`CSL_Status`

- `CSL_SOK` - Successful on getting the base address of CFG
- `CSL_ESYS_FAIL` - The CFG instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid parameter.

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure

Example

```
CSL_Status           status;
CSL_CfgBaseAddress   baseAddress;
...
status = CSL_cfgGetBaseAddress(   CSL_MEMPROT_CONFIG,
                                  NULL,
                                  &baseAddress);
...
```

20.3 Data Structures

This section lists the data structures available in the CFG module.

20.3.1 CSL_CfgObj

Detailed Description

This object contains the reference to the instance of CFG opened using the *CSL_cfgOpen()*. The pointer to this is passed as CFG Handle to all CFG CSL APIs. CSL_cfgOpen() function initializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_CfgObj::cfgNum

This is the instance of CFG being referred to by this object

CSL_CfgRegsOvly CSL_CfgObj::regs

This is a pointer to the registers of the instance of CFG referred to by this object

20.3.2 CSL_CfgFaultStatus

Detailed Description

CSL_CfgStatus has all the fields required for the status information of CFG module.

Field Documentation

CSL_BitMask16 CSL_CfgFaultStatus::errorMask

Bit Mask of the Errors

Uint16 CSL_CfgFaultStatus::faultId

Fault Id. The IDof the originator of the faulting access

20.3.3 CSL_CfgContext

Detailed Description

Cfg specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_CfgContext::contextInfo

Context information of Cfg CSL passed as an argument to CSL_cfgInit(). Present implementation of Cfg CSL doesn't have any context information; hence assigned NULL. The declaration is just a placeholder for future implementation.

20.3.4 CSL_CfgParam

Detailed Description

This is module specific parameter. Present implementation of Cfg CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_CfgParam::flags

Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation. Passed as an argument to CSL_cfgOpen().

20.3.5 CSL_CfgBaseAddress

Detailed Description
This structure contains the base address information for the Cfg instance.

Field Documentation

CSL_CfgRegsOvly CSL_CfgBaseAddress::regs
Base address of the configuration registers of the peripheral

20.4 Enumerations

This section lists the enumerations available in the CFG module.

20.4.1 CSL_CfgHwControlCmd

enum CSL_CfgHwControlCmd
Enumeration for queries passed to *CSL_cfgHwControl()*.
This is used to select the commands to control the operations existing setup of CFG. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_CFG_CMD_CLEAR	CFG Hardware control command to clears the error conditions stored in MPFAR and MPFSR. **Parameters:** *None*

20.4.2 CSL_CfgHwStatusQuery

enum CSL_CfgHwStatusQuery
Enumeration for queries passed to *CSL_cfgGetHwStatus()*.
This is used to get the status of different operations or to get the existing setup of CFG.

Enumeration values:

CSL_CFG_QUERY_FAULT_ADDR	Status query command to get the Fault Address. **Parameters:** *(Uint32 *)*
CSL_CFG_QUERY_FAULT_STATUS	Status query command to get the Status information of CSL_CfgStatus. **Parameters:** *(CSL_CfgFaultStatus *)*

20.5 Macros

#define CSL_CFG_FAULT_STAT_FID (0x0000F700u)
Mask value of Fault ID

#define CSL_CFG_FAULT_STAT_LOCAL (0x00000080u)
Mask value to get the status of Local memory (L1/L2)

#define CSL_CFG_FAULT_STAT_SR (0x00000020u)
Mask value for Supervisor Read

#define CSL_CFG_FAULT_STAT_SW (0x00000010u)
Mask value for Supervisor Write

#define CSL_CFG_FAULT_STAT_SX (0x00000008u)
Mask value for Supervisor Execute

#define CSL_CFG_FAULT_STAT_UR (0x00000004u)
Mask value for User Read

#define CSL_CFG_FAULT_STAT_UW (0x00000002u)
Mask value for User Write

#define CSL_CFG_FAULT_STAT_UX (0x00000001u)
Mask value for User Execute

20.6 Typedefs

typedef CSL_CfgObj * CSL_CfgHandle
This is a pointer to CSL_CfgObj & is passed as the first parameter to all CFG CSL APIs

Chapter 21
CHIP Module

Topics

21.1 Overview

This module deals with all System On Chip (SOC) configurations. It constitutes of Configuration Registers specific for the chip.

Following are the Registers associated with the CHIP module:

- Addressing Mode Register - This register specifies the addressing mode for the registers which can perform linear or circular addressing, also contain sizes for circular addressing
- Control Status Register - This register contains the control and status bits. This register is used to control the mode of cache. This is also used to enable or disable all the interrupts except reset and non maskable interrupt.
- Interrupt Flag Register – This register contains the status of INT4−INT15 and NMI interrupt. Each corresponding bit in the IFR is set to 1 when that interrupt occurs; otherwise, the bits are cleared to 0.
- Interrupt Set Register - This register allows user to manually set the maskable interrupts (INT4−INT15) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ISR causes the corresponding interrupt flag to be set in IFR.
- Interrupt Clear Register – This register allows user to manually clear the maskable interrupts (INT15−INT4) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ICR causes the corresponding interrupt flag to be cleared in IFR.
- Interrupt Enable Register - This register enables and disables individual interrupts and this not accessible in User mode.
- Interrupt Service Table Pointer Register – This register is used to locate the interrupt service routine (ISR).
- Interrupt Return Pointer Register – This register contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt.
- Nonmaskable Interrupt (NMI) Return Pointer Register – This register contains the return pointer that directs the CPU to the proper location to continue program execution after processing of a non−maskable interrupt (NMI).
- Exception Return Pointer Register – This register contains the return pointer that directs the CPU to the proper location to continue program execution after processing of a exception.
- Time Stamp Counter Registers – The CPU contains a free running 64-bit counter that advances each CPU clock after counting is enabled. The counter is accessed using two 32-bit read-only control registers, Time Stamp Counter Registers – Low (TSCL) and Time Stamp Counter Registers – High (TSCH). The counter is enabled by writing to TSCL. The value written is ignored. Once enabled, counting cannot be disabled under program control. Counting is disabled in the following cases:
    - After exiting the reset state.
    - When the CPU is fully powered down.
- SPLOOP Inner Loop Count Register - The SPLOOP or SPLOOPD instructions use the SPLOOP inner loop count register (ILC), as the count of the number of iterations left to perform. The ILC content is decremented at each stage boundary until the ILC content reaches 0.
- SPLOOP Reload Inner Loop Count Register - Predicated SPLOOP or SPLOOPD instructions used in conjunction with a SPMASKR or SPKERNELR instruction use the SPLOOP reload inner loop count register (RILC), as the iteration count value to be written to the SPLOOP inner loop count register (ILC) in the cycle before the reload operation begins.
- E1 Phase Program Counter – This register contains the 32-bit address of the fetch packet in the E1 pipeline phase.

- DSP Core Number Register – This register provides an identifier to shared resources in the system which identifies which CPU is accessing those resources. The contents of this register are set to a specific value at reset.
- Saturation Status Register – This register provides saturation flags for each functional unit, making it possible for the program to distinguish between saturations caused by different instructions in the same execute packet.
- GMPY Polynomial.A Side Register – The GMPY instruction uses the 32-bit polynomial in the GMPY polynomial—A side register (GPLYA), when the instruction is executed on the M1 unit.
- GMPY Polynomial. B Side Register – The GMPY instruction uses the 32-bit polynomial in the GMPY polynomial—B side register (GPLYB), when the instruction is executed on the M2 unit.
- Galois Field Polynomial Generator Function Register – This register controls the field size and the Galois field polynomial generator of the Galois field multiply hardware.
- Task State Register – This register contains all of the status bits that determine or indicate the current execution environment. TSR is saved in the event of an interrupt or exception to the ITSR or NTSR, respectively.
- Interrupt Task State Register – This register is used to store the contents of the task state register (TSR) in the event of an interrupt.
- NMI/Exception Task State Register – This register is used to store the contents of the task state register (TSR) and the conditions under which an an exception occurred in the event of a nonmaskable interrupt (NMI) or an exception.
- Exception Flag Register – This register contains bits that indicate which exceptions have been detected. Clearing the EFR bits is done by writing a 1 to the corresponding bit position in the exception clear register (ECR).
- Exception Clear Register – This register is used to clear individual bits in the exception flag register (EFR). Writing a 1 to any bit in ECR clears the corresponding bit in EFR.
- Internal Exception Report Register – This register contains flags that indicate the cause of the internal exception.
- Restricted Entry Point Address Register – This register is used by the SWENR instruction as the target of the change of control when an SWENR instruction is issued. The contents of REP should be preinitialized by the processor in Supervisor mode before any SWENR instruction is issued.

21.2 Functions

This section lists the functions available in the CHIP module.

21.2.1 CSL_chipWriteReg

Uint32 CSL_chipWriteReg (CSL_ChipReg *reg*, CSL_Reg32 *val*)

Description
This API writes specified control register with the specified value 'val'. The register that can be specified could be one of those enumerated in CSL_ChipReg.

Arguments

`reg`	`This is the register name specified for the register through the enum`
`val`	`Value to be written into the register`

Return Value
`Uint32`

The value in the register before the new value being written

- Old programmed value

Pre Condition
None

Post Condition
The reg control register is written with the value passed.

Modifies
The specified register will be modified.

Usage Constraints
Please refer to the C64x+ user guide for constraints while accessing registers in different privilege levels

Example

```
Uint32 oldVal;
oldVal = CSL_chipWriteReg(CSL_CHIP_AMR, 56);
...
```

21.2.2 CSL_chipReadReg

Uint32 CSL_chipReadReg (CSL_ChipReg *reg*)

Description
This API reads the specified control register. The register that can be specified could be one of those enumerated in CSL_ChipReg.

Arguments

```
reg      This is the register name specified for the register
         through the enum
```

Return Value
`Uint32`

- The value read from the register

Pre Condition
None

Post Condition
None

Modifies
None

Usage Constraints
Please refer to the C64x+ user guide for constraints while accessing registers in different privilege levels

Example

```
Uint32 regVal;
regVal = CSL_chipReadReg(CSL_CHIP_AMR);
...
```

21.3 Enumerations

This section lists the enumerations available in the CHIP module.

21.3.1 CSL_ChipReg

enum CSL_ChipReg
Enumeration for the CHIP registers

Enumeration values:

CSL_CHIP_AMR	Addressing Mode Register
CSL_CHIP_CSR	Control Status Register
CSL_CHIP_IFR	Interrupt Flag Register
CSL_CHIP_ISR	Interrupt Set Register
CSL_CHIP_ICR	Interrupt Clear Register
CSL_CHIP_IER	Interrupt Enable Register
CSL_CHIP_ISTP	Interrupt Service Table Pointer Register
CSL_CHIP_IRP	Interrupt Return Pointer Register
CSL_CHIP_NRP	Nonmaskable Interrupt (NMI) Return Pointer Register
CSL_CHIP_ERP	Exception Return Pointer Register
CSL_CHIP_TSCL	Time Stamp Counter Register - Low
CSL_CHIP_TSCH	Time Stamp Counter Registers - High
CSL_CHIP_ILC	SPLOOP Inner Loop Count Register
CSL_CHIP_RILC	SPLOOP Reload Inner Loop Count Register
CSL_CHIP_REP	Restricted Entry Point Address Register
CSL_CHIP_PCE1	E1 Phase Program Counter
CSL_CHIP_DNUM	DSP Core Number Register
CSL_CHIP_SSR	Saturation Status Register
CSL_CHIP_GPLYA	GMPY Polynomial A Side Register
CSL_CHIP_GPLYB	GMPY Polynomial B Side Register
CSL_CHIP_GFPGFR	Galois Field Polynomial Generator Function Register
CSL_CHIP_TSR	Task State Register
CSL_CHIP_ITSR	Interrupt Task State Register
CSL_CHIP_NTSR	NMI/Exception Task State Register
CSL_CHIP_EFR	Exception Flag Register
CSL_CHIP_ECR	Exception Clear Register
CSL_CHIP_IERR	Internal Exception Report Register

Chapter 22
IDMA MODULE

Topics

22.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within IDMA module.

The internal DMA (IDMA), is a DMA local to the megamodule- that is, it provides data move services only within the megamodule (L1P, L1D, L2, and CFG).
There are two IDMA channels (0 and 1).

- Channel 0 allows data to be transferred between the peripheral configuration space (CFG) and any local memories (L1P, L1D, and L2).
- Channel 1 is used to transfer data between the local memories (L1P, L1D, and L2).

The IDMA data transfers occur in the background of CPU operation. That is, once a channel transfer is programmed, it happens concurrent with other CPU activity, and without additional CPU intervention.

22.2 Functions

This section lists the functions available in the IDMA module.

22.2.1 IDMA1_init

Int IDMA1_init ([IDMA_priSet](#) ***priority***,
[IDMA_intEn](#) ***interr***
)

Description
This function obtains a priority and an interrupt flag and remembers them so that all future transfers on channel 1 will use these priorities. The priority is contained in the argument "priority" and interrupt flag in "interr". This function performs IDMA Channel 1 initialization by setting the priority level and the enabling/disabling the interrupt event generation for the channel.

Arguments

```
priority        Priority 0-7 of handle

interr          Interrupt event generated on/off
```

Return Value
`Int`

- Priority of IDMA relative to CPU and whether interrupt is desired or not. These values stored in the 32-bit field 'cnt' of the local IDMA1 handle structure

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Int         cnt1;

// Initialize IDMA Channel 1
// Set Chan 1 to have Priority 7 and Interrupt Event Gen On
...

cnt1 = IDMA1_init(IDMA_PRI_7, IDMA_INT_EN);
...
```

22.2.2 IDMA1_copy

Int IDMA1_copy (**Uint32 *** ***src***,
Uint32 * ***dst***,
Uint32 ***byteCnt***

)

Description
IDMA1_copy() transfers "byteCnt" bytes from a source "src" to a destination "dst". It is assumed that both the source and destination addresses are in internal memory. Transfers from addresses that are not in the internal memory will raise an exception. No checking is performed by this function to check the correctness of any of the passed in arguments. Used to transfer "byteCnt" bytes from source "src" to destination "dst".

Arguments

`src`	`Pointer to the source address`
`dst`	`Pointer to the destination address`
`byteCnt`	`Number of bytes to be transferred`

Return Value
`Int`

- Always returns 0

Pre Condition
The function *IDMA1_init()* must be called successfully before calling to this function.

Post Condition
Call *IDMA1_wait ()* function

Modifies
The hardware registers of IDMA.

Example
```
#pragma DATA_SECTION    (src, "ISRAM");
#pragma DATA_ALIGN      (src, 8);
#pragma DATA_SECTION    (dst1, "ISRAM1");
#pragma DATA_ALIGN      (dst1, 8);
Uint32        src[20] =
{
      0xDEADBEEF, 0xFADEBABE, 0x5AA51C3A, 0xD4536BA3,
      0x5E69BA23, 0x4884A01F, 0x9265ACDA, 0xFFFF0123,
      0xBEADDABE, 0x234A76B2, 0x9675ABCD, 0xABCDEF12,
      0xEEEECDEA, 0x01234567, 0x00000000, 0xFEEDFADE,
      0x0A1B2C3D, 0x4E5F6B7C, 0x5AA5ECCE, 0xFABEFACE
};
Uint32        dst1[20];
// Copy src to dst1 - 80 bytes - 20 words
IDMA1_copy(src, dst1, 80);
...
```

22.2.3 IDMA1_fill

Int IDMA1_fill	**(**	**Uint32 ***	***dst*,**
		Uint32	***byteCnt*,**
		Uint32	***fill_value***

)

Description
IDMA1_fill() takes a fill value in "fill_value" and fills "byteCnt" bytes of the "fill_value" to destination "dst".

Arguments

```
dst          Pointer to the destination address

byteCnt      Number of bytes to be transferred

fill_value   Data to be filled
```

Return Value
`Int`

- Always returns 0

Pre Condition
The function *IDMA1_init()* must be called successfully before calling to this function.

Post Condition
Call *IDMA1_wait()* function

Modifies
The hardware registers of IDMA

Example

```
#pragma DATA_SECTION  (dst1, "ISRAM1");
#pragma DATA_ALIGN    (dst1, 8);

Uint32        dst1[20];
IDMA1_fill(dst1, 80, 0xAAAABABA);
...
```

22.2.4 IDMA1_getStatus

Uint32 IDMA1_getStatus (void)

Description
IDMA1_getStatus() gets the active and pending status of IMDA Channel 1 and returns ACTV in the least significant bit and PEND in the 2nd least significant bit

Arguments
None

Return Value
`Uint32`

- IDMA channel 1 status

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32        stat;
stat = IDMA1_getStatus();
...
```

22.2.5 IDMA1_wait

void IDMA1_wait (void)

Description
IDMA1_wait() waits until all previous transfers for IDMA Channel 1 have been completed by making sure that both active and pending bits are zero. These are the two least significant bits of the status register for the channel.

Arguments
None

Return Value
None

Pre Condition
Functions IDMA1_init() and IDMA1_copy() or IDMA1_fill() must be called successfully in order before calling this API.

Post Condition
Completion of previous transfers

Modifies
IDMA channel 1 registers

Example

```
#pragma DATA_SECTION  (dst, "ISRAM1");
#pragma DATA_ALIGN    (dst, 8);
Uint32        dst[20];
Uint32        stat;
...
IDMA_fill(dst, 80, 0xAAAAAAAA);
stat = IDMA1_getStatus();
IDMA1_wait();
...
```

22.2.6 IDMA1_setPriority

Int IDMA1_setPriority (IDMA_priSet *priority*)

Description
IDMA1_setPriority() sets a "3-bit" priority field which has a valid range of 0-7 for priorities 0-7. It

returns a "32-bit" count register field back to the user. This 32-bit register field will be used in IDMA1_copy() and IDMA1_fill() to program the Priority and Interrupt options for IDMA Chan 1 Sets the priority level for IDMA channel 1 transfers.

Arguments

```
priority        Priority 0-7 of handle
```

Return Value

`Int`

- Priority of IDMA relative to CPU. This value stored in the 32-bit field 'cnt' of the local IDMA1 handle structure

Pre Condition

None

Post Condition

None

Modifies

IDMA channel 1 registers

Example

```
Uint32        tempCnt;
...

// Set and test Priority level for IDMA1
tempCnt = IDMA1_setPriority(IDMA_PRI_2);
...
```

22.2.7 IDMA1_setInt

Int IDMA1_setInt (IDMA_intEn *interr*)

Description

IDMA1_setInt() sets the interrupt enable field which is used to enable/disable interrupts for IDMA Channel 1. It returns a "32-bit" count register field back to the user. This 32-bit register field will be used in IDMA1_copy() and IDMA1_fill() to program the Priority and Interrupt options for IDMA Channel 1.

Arguments

```
interr        Interrupt event generated on/off
```

Return Value

`Int`

- Interrupt is enabled or not. This value stored in the 32-bit field 'cnt' of the local IDMA1 handle structure

Pre Condition

None

Post Condition

None

Modifies
IDMA channel 1 registers

Example

```
Uint32        tempCnt;
...

// Set and test Interrupt event gen for IDMA1
tempCnt = IDMA1_setInt(IDMA_INT_DIS);
...
```

22.2.8 IDMA0_init

Int IDMA0_init (IDMA_intEn *interr*)

Description
This function obtains a interrupt enable setting and remembers them so that all the future transfers on Channel 0 generate interrupts or not. Initializes the Interrupt Event Generation for IDMA Channel 0.

Arguments

```
interr        Interrupt event generated on/off
```

Return Value
`Int`

- Interrupt is enabled or not.This value stored in the 32-bit 'cnt' field of the local IDMA0 configuration structure

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32        cnt0;
...
// Initialize IDMA Channel 0
// Set Chan 0 to have Interrupt Event Gen On
cnt0 = IDMA0_init(IDMA_INT_EN);
...
```

22.2.9 IDMA0_config

void IDMA0_config (IDMA0_Config * *config*)

Description
IDMA0_config() - Configures IDMA Channel 0 to perform a transfer between Internal Memory and Configuration Space based on the data in the *config structure. "mask" provides a 1-hot encoding for the 32-word transfer that determines which of the 32-words are to be transferred. In the *config structure "src" provides the source location of the transfer and "dst provides the destination location of the transfer and both must be word aligned. While "cnt" provides the number of 32-word transfers to perform and must not be greater than 15. Initializes the configuration for IDMA Channel 0 including 1-hot encoding mask, source location, destination location and count. This is done using the structure IDMA0_Config.

Arguments

```
config        Pointer to the Configuration structure
```

Return Value
None

Pre Condition
The function IDMA0_init() must be called successfully before invoking this API.

Post Condition
Invoke IDMA0_wait() after calling this API

Modifies
The hardware registers of IDMA.

Example

```
IDMA0_Config   config
...
IDMA0_config(&config);
IDMA0_wait();
...
```

22.2.10 IDMA0_configArgs

void IDMA0_configArgs	**(**	**Uint32**	***mask*,**
		Uint32 *	***src*,**
		Uint32 *	***dst*,**
		Uint32	***count***
	)		

Description
IDMA0_configArgs() - Configures IMDA Channel 0 to perform a transfer between Internal Memory and Configuration Space based on the inputs to the function. "mask" provides a 1-hot encoding for the 32-word transfer that determines which of the 32-words are to be transferred. "src" provides the source location of the transfer and "dst provides the destination location of the transfer and both must be word aligned. While "cnt" provides the number of 32-word transfers to perform and must not be greater than 15. Initializes the configuration for IDMA Channel 0 including 1-hot encoding mask, source location, destination location and count.

Arguments

```
mask        Encoding value for the 32-word transfer

src         Pointer to the source location of the transfer

dst         Pointer to the destination location of the transfer

count       Number of 32-word transfers
```

Return Value
None

Pre Condition
The function IDMA0_init() must be called successfully before invoking this API.

Post Condition
Invoke IDMA0_wait() after calling this API

Modifies
The hardware registers of IDMA.

Example

```
Uint32  src,dst;
Uint32  mask;
...
IDMA0_configArgs(mask, src, dst, 1);
IDMA0_wait();
...
```

22.2.11 IDMA0_getStatus

Uint32 IDMA0_getStatus (void)

Description
IDMA0_getStatus() gets the active and pending status of IMDA Channel 0 and returns ACTV in the least significant bit and PEND in the 2nd least significant bit.

Arguments
None

Return Value
`Uint32`

- IDMA channel 0 status

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
Uint32        stat;
...
stat = IDMA0_getStatus();
...
```

22.2.12 IDMA0_wait

void IDMA0_wait (void)

Description
IDMA0_wait() waits until all previous transfers for IDMA Channel 0 have been completed by making sure that both active and pend bits are zero. These are the two least significant bits of the status register for the channel.

Arguments
None

Return Value
None

Pre Condition
Functions IDMA0_init() and IDMA0_config() or IDMA0_configArgs () must be called successfully in order before calling this API.

Post Condition
Completion of previous transfer

Modifies
IDMA channel 0 registers

Example

```
Uint32        stat;
...
stat = IDMA0_getStatus();
IDMA0_wait();
...
```

22.2.13 IDMA0_setInt

Int IDMA0_setInt (IDMA_intEn *interr*)

Description
IDMA0_setInt() sets a the interrupt enable field which is used to enable/disable interrupts for IDMA Channel 0. It returns a "32-bit" count register field back to the user. This 32-bit register field will be used in IDMA0_config() and IDMA0_configArgs() to program the Interrupt option for IDMA Channel 0

Arguments

```
interr        Interrupt event generated on/off
```

Return Value

`Int`

- Interrupt is enabled or not. This value stored in the 32-bit ‘cnt’ field of the local IDMA0 configuration structure

Pre Condition

None

Post Condition

None

Modifies

IDMA channel 0 registers

Example

```
Uint32        tempCnt;
...
// Set and test Interrupt event gen for IDMA0
tempCnt = IDMA0_setInt(IDMA_INT_DIS);
...
```

22.3 Data Structures

This section lists the data structures available in the IDMA module.

22.3.1 idma1_handle

Detailed Description

IDMA1_handle IDMA Channel 1 handle - Contains Status, Source and Destination locations and count for channel 1 transfer.

Field Documentation

Uint32 idma1_handle::cnt

Number of bytes to be transferred

Uint32* idma1_handle::dst

IDMA channel 1 destination

Uint32 idma1_handle::reserved

Reserved area

Uint32* idma1_handle::src

IDMA channel 1 source location

Uint32 idma1_handle::status

IDMA channel 1 status

22.3.2 idma0_config

Detailed Description

IDMA0_Config IDMA Channel 0 configuration - Contains Status, Mask, Source and Destination locations and count for channel 0 (configuration) transfers.

Field Documentation

Uint32 idma0_config::cnt

Number of bytes to be transferred

Uint32* idma0_config::dst

IDMA channel 0 destination

Uint32 idma0_config::mask

IDMA channel 0 mask value

Uint32* idma0_config::src

IDMA channel 0 source location

Uint32 idma0_config::status

IDMA channel 0 status

22.4 Enumerations

This section lists the enumerations available in the IDMA module.

22.4.1 IDMA_Chan

enum IDMA_Chan
This enumeration specifies which IDMA channel will be used. This is used to indicate which IDMA channel (0 or 1) will be used by API.

Enumeration values:

IDMA_CHAN_0	IDMA channel 0
IDMA_CHAN_1	IDMA channel 1

22.4.2 IDMA_intEn

enum IDMA_intEn
This enumeration specifies whether the interrupt event generation is enabled or disabled. This is used to indicate whether the interrupt event generation is enabled or disabled.

Enumeration values:

IDMA_INT_DIS	Idma Int Disable
IDMA_INT_EN	Idma Int Enable

22.4.3 IDMA_priSet

enum IDMA_priSet
This enumeration specifies what priority level the IDMA channel is set to. This is used to specify what priority level the IDMA channel is set to.

Enumeration values:

IDMA_PRI_0	Set Priority level 0
IDMA_PRI_1	Set Priority level 1
IDMA_PRI_2	Set Priority level 2
IDMA_PRI_3	Set Priority level 3
IDMA_PRI_4	Set Priority level 4
IDMA_PRI_5	Set Priority level 5
IDMA_PRI_6	Set Priority level 6
IDMA_PRI_7	Set Priority level 7
IDMA_PRI_NULL	No Priority level

22.5 Typedefs

typedef CSL_IdmaRegs* CSL_idmaOvly
Pointer to the register overlay structure of the IDMA

typedef struct idma0_config IDMA0_Config
IDMA0_Config IDMA Channel 0 configuration - Contains Status, Mask, Source and Destination locations and count for channel 0 (configuration) transfer.

typedef struct idma1_handle IDMA1_handle
IDMA1_handle IDMA Channel 1 handle - Contains Status, Source and Destination locations and count for channel 1 transfer.

typedef Uint32 Status
Status

Chapter 23
MEMPROT Module

Topics

23.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within MEMPROT module

Memory protection used to support resources (L1P, L2, L1D not an Intenal CFG space). Memory protection provides many benefits to a system.

Memory protection functionality can:

- Protect operating system data structures from poorly behaving code.
- Aid in debugging by providing greater information about illegal memory accesses.
- Allow the operating system to enforce clearly defined boundaries between supervisor and user modeaccesses, leading to greater system robustness.

23.2 Functions

This section lists the functions available in the MEMPROT module.

23.2.1 CSL_memprotInit

CSL_Status CSL_memprotInit (CSL_MemprotContext * *pContext*)

Description
This is the initialization function for the MEMPROT CSL. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext   Context information for the instance. Should be NULL
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
CSL_Status      status;
...
status = CSL_memprotInit(NULL);
...
```

23.2.2 CSL_memprotOpen

CSL_MemprotHandle CSL_memprotOpen (CSL_MemprotObj * *pMemprotObj*, CSL_InstNum *memprotNum*, CSL_MemprotParam * *pMemprotParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the instance and returns a handle to the MEMPROT instance. The open call sets up the data structures for the particular instance of MEMPROT device.The device can be re-opened anytime after it has been normally closed, if so

required. The handle returned by this call is input as an essential argument for rest of the APIs described for this module.

Arguments

`pMemprotObj`	`Pointer to the MEMPROT instance object`
`memprotNum`	`Instance of the MEMPROT to be opened`
`pMemprotParam`	`Pointer to module specific parameters`
`pStatus`	`Pointer for returning status of the function call`

Return Value
CSL_MemprotHandle

- Valid MEMPROT instance handle will be returned if status value is equal to CSL_SOK.

Pre Condition
Memory protection must be successfully initialized via CSL_memprotInit() before calling this function. Memory for the CSL_MemprotObj must be allocated outside this call. This object must be retained while usage of this module. Depending on the module opened some inherant constraints need to be kept in mind. When a handle for the Config block is opened the only operation possible is a query for the fault Status. No other control command/ query/ setup must be used. When a handle for L1D/L1P is opened, then the constraints with respect to the number of Memory pages must be kept in mind.

Post Condition
1. MEMPROT object structure is populated
2. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid MEMPROT module handle is returned
- `CSL_ESYS_FAIL` - The MEMPROT instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Modifies
1. The status variable
2. MEMPROT object structure

Example

```
CSL_MemprotObj    mpL2Obj;
CSL_MemprotHandle hmpL2;
CSL_Status        status;
// Initializing the module
CSL_memprotInit(NULL);

// Opening the Handle for the L2
hmpL2 = CSL_memprotOpen(&mpL2Obj,
                        CSL_MEMPROT_L2,
                        NULL,
                        &status);
...
```

23.2.3 CSL_memprotClose

CSL_Status CSL_memprotClose (**CSL_MemprotHandle** ***hMemprot***)

Description
This function closes the specified instance of MEMPROT.

Arguments

```
hMemprot        Handle to the MEMPROT instance
```

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both CSL_memprotInit() and CSL_memprotOpen() must be called successfully in that order before this function can be called

Post Condition
The MEMPROT CSL APIs can not be called until the MEMPROT CSL is reopened again using CSL_memprotOpen().

Modifies
CSL_memprotObj structure values

Example

```
CSL_MemprotHandle   hMemprot;
CSL_Status          status;

...
status = CSL_memprotClose(hMemprot);
...
```

23.2.4 CSL_memprotHwSetup

CSL_Status CSL_memprotHwSetup (**CSL_MemprotHandle** ***hMemprot***, **CSL_MemprotHwSetup** * ***setup***)

Description
This function initializes the module registers with the appropriate values provided through the HwSetup Data structure. For information passed through the HwSetup data structure, refer CSL_memprotHwSetup.

Arguments

```
hMemprot      Handle to the memprot instance
```

```
setup         Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Hardware structure is not properly initialized

Pre Condition
Both CSL_memprotInit() and CSL_memprotOpen() must be called successfully in order before calling this function. The user has to allocate space for & fill in the main setup structure appropriately before calling this function. Ensure numpages is not set to > 32 for handles for L1D/L1P. Ensure numpages is not > 64 for L2.

Post Condition

MEMPROT registers are configured according to the hardware setup parameters

Modifies
The hardware registers of MEMPROT.

Example

```
#define PAGE_ATTR    0xFFF0

CSL_MemprotObj       mpL2Obj;
CSL_MemprotHandle    hmpL2;
CSL_Status           status;
CSL_MemprotHwSetup   L2MpSetup;
Uint16 pageAttrTable[10] = {PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR};
Uint32 key[2] = {0x11223344,0x55667788};
// Initializing the module
CSL_memprotInit(NULL);

// Opening the Handle for the L2
hmpL2 = CSL_memprotOpen(&mpL2Obj, CSL_MEMPROT_L2, NULL, &status);
L2MpSetup. memPageAttr = pageAttrTable;
L2MpSetup.numPages = 10;
L2MpSetup.key = key;

// Do Setup for the L2 Memory protection/
CSL_memprotHwSetup(hmpL2, &L2MpSetup);
...
```

23.2.5 CSL_memprotGetHwSetup

CSL_Status CSL_memprotGetHwSetup **(** **[CSL_MemprotHandle](#)** ***hMemprot*****,**
[CSL_MemprotHwSetup](#) * ***setup***
)

Description
This function gets the current setup of the Memory Protection registers. The status is returned through CSL_MemprotHwSetup. The obtaining of status is the reverse operation of CSL_MemprotHwSetup() function. Only the Memory Page attributes are read and filled into the HwSetup structure.

Arguments

```
hMemprot         Handle to the MEMPROT instance

setup            Pointer to setup structure which contains the
                 setup information of MEMPROT.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Setup info load successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both CSL_memprotInit() andCSL_memprotOpen() must be called successfully in order before calling CSL_memprotGetHwSetup(). Ensure numpages is initialized depending on the number of desired attributes in the setup. Make sure to set numpages <= 32 for handles for L1D/L1P. Ensure numpages <= 64 for L2.

Post Condition
None

Modifies
Second parameter setup value

Example

```
#define PAGE_ATTR 0xFFF0

CSL_MemprotObj       mpL2Obj;
CSL_MemprotHandle    hmpL2;
CSL_Status           status;
CSL_MemprotHwSetup   L2MpSetup,L2MpGetSetup;
Uint16 pageAttrTable[10] = {PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR};
Uint32 key[2] = {0x11223344,0x55667788};

// Initializing the module
```

```
CSL_memprotInit(NULL);

// Opening the Handle for the L2
hmpL2 = CSL_memprotOpen(&mpL2Obj,CSL_MEMPROT_L2,NULL,&status);
L2MpSetup. memPageAttr = pageAttrTable;
L2MpSetup.numPages = 10;
L2MpSetup.key = key;

// Do Setup for the L2 Memory protection/
CSL_memprotHwSetup(hmpL2,&L2MpSetup);
status = CSL_memprotGetHwSetup(hmpL2,&L2MpGetSetup);
...
```

23.2.6 CSL_memprotHwControl

CSL_Status CSL_memprotHwControl	**(**	**CSL_MemprotHandle**	***hMemprot*, **
		CSL_MemprotHwControlCmd	***cmd*, **
		void *	***arg***
	)		

Description
Control operations for the Memory protection registers. For a particular control operation, the pointer to the corresponding data type needs to be passed as argument HwControl function call. All the arguments (structure elements included) passed to the HwControl function are inputs. For the list of commands supported and argument type that can be *void** casted & passed with a particular command refer to CSL_MemprotHwControlCmd.

Arguments

`hMemprot`	`Handle to the MEMPROT instance`
`cmd`	`The command to this API indicates the action to be taken on MEMPROT.`
`arg`	`An optional argument`

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_FAIL` - Invalid lock status
- `CSL_ESYS_INVPARAMS` - Invalid Parameter

Pre Condition
Both CSL_memprotInit() and CSL_memprotOpen() must be called successfully in order before calling CSL_memprotHwControl(). For the argument type that can be void* casted and passed with a particular command refer to CSL_MemprotHwControlCmd.

Post Condition

MEMPROT registers are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
The hardware registers of MEMPROT.

Example

```
#define PAGE_ATTR 0xFFF0

Uint16 pageAttrTable[10] = {PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR};
Uint32 key[2] = {0x11223344,0x55667788};

CSL_MemprotObj             mpL2Obj;
CSL_MemprotHandle          hmpL2;
CSL_Status                 status;
CSL_MemprotHwSetup         L2MpSetup;
CSL_MemprotLockStatus      lockStat;


// Initializing the module
CSL_memprotInit(NULL);

// Opening the Handle for the L2
hmpL2 = CSL_memprotOpen(&mpL2Obj,CSL_MEMPROT_L2,NULL,&status);
L2MpSetup. memPageAttr = pageAttrTable;
L2MpSetup.numPages = 10;
L2MpSetup.key = key;

// Do Setup for the L2 Memory protection/
CSL_memprotHwSetup(hmpL2,&L2MpSetup);

// Query Lock Status
CSL_memprotGetHwStatus(hmpL2,CSL_MEMPROT_QUERY_LOCKSTAT,&lockStat);
// Unlock the Unit if Locked
if (lockStat == CSL_MEMPROT_LOCKSTAT_LOCK) {
    status = CSL_memprotHwControl(hmpL2,CSL_MEMPROT_CMD_UNLOCK,key);
}
...
```

23.2.7 CSL_memprotGetHwStatus

CSL_Status CSL_memprotGetHwStatus	**(**	**CSL_MemprotHandle**	***hMemprot*,**
		CSL_MemprotHwStatusQuery	***query*,**
		void *	***response***
	)		

Description
This function is used to read the current module configuration, status flags and the value present associated registers. User should allocate memory for the said data type and pass its pointer as

an unadorned void* argument to the status query call. For details about the various status queries supported and the associated data structure to record the response, refer to CSL_MemprotHwStatusQuery.

Arguments

`hMemprot`	`Handle to the MEMPROT instance`
`query`	`The query to this API of MEMPROT which indicates the status to be returned.`
`response`	`Placeholder to return the status.`

Return Value
`CSL_Status`

- `CSL_SOK` - Status info return successful.
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - Invalid query
- `CSL_ESYS_INVPARAMS` - Invalid parameter

Pre Condition
Both CSL_memprotInit() and CSL_memprotOpen() must be called successfully in order before calling CSL_memprotGetHwStatus(). For the argument type that can be void* casted and passed with a particular command refer to CSL_MemprotHwStatusQuery.

Post Condition
None

Modifies
Third parameter “response” value

Example

```
#define PAGE_ATTR 0xFFF0

Uint16 pageAttrTable[10] = {PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR,
                            PAGE_ATTR,PAGE_ATTR,PAGE_ATTR};
Uint32 key[2] = {0x11223344,0x55667788};
CSL_MemprotObj             mpL2Obj;
CSL_MemprotHandle          hmpL2;
CSL_Status                 status;
CSL_MemprotHwSetup         L2MpSetup;
CSL_MemprotLockStatus      lockStat;

// Initializing the module
CSL_memprotInit(NULL);

// Opening the Handle for the L2
hmpL2 = CSL_memprotOpen(&mpL2Obj,CSL_MEMPROT_L2,NULL,&status);
L2MpSetup. memPageAttr = pageAttrTable;
L2MpSetup.numPages = 10;
L2MpSetup.key = key;
```

```
// Do Setup for the L2 Memory protection/
CSL_memprotHwSetup(hmpL2,&L2MpSetup);

// Query Lock Status
CSL_memprotGetHwStatus(hmpL2,CSL_MEMPROT_QUERY_LOCKSTAT,&lockStat);
...
```

23.2.8 CSL_memprotGetBaseAddress

CSL_Status CSL_memprotGetBaseAddress(CSL_InstNum ***memprotNum*****,**
CSL_MemprotParam * ***pMemprotParam*****,**
CSL_MemprotBaseAddress * ***pBaseAddress***
)

Description
Function to get the base address of the peripheral instance. This function is used for getting the base address of the peripheral instance. This function will be called inside the CSL_memprotOpen() function call. This function is open for re-implementing if the user wants to modify the base address of the peripheral object to point to a different location and there by allow CSL initiated write/reads into peripheral. MMRs go to an alternate location.

Arguments

```
memprotNum      Specifies the instance of the memprot to be
                opened.

pMemprotParam   Module specific parameters.

pBaseAddress    Pointer to base address structure containing base
                address details.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Successfull on getting the base address of MEMPROT.
- `CSL_ESYS_FAIL` - The instance number is invalid.
- `CSL_ESYS_INVPARAMS` - Invalid parameter.

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status              status;
```

```
CSL_MemprotBaseAddress  baseAddress;

...
status = CSL_memprotGetBaseAddress(CSL_MEMPROT_L2, NULL,
                                   &baseAddress);
...
```

23.3 Data Structures

This section lists the data structures available in the MEMPROT module.

23.3.1 CSL_MemprotObj

Detailed Description

This object contains the reference to the instance of memory Protection Module opened using the CSL_memprotOpen(). A pointer to this object is passed to all Memory Protection CSL APIs.

Field Documentation

CSL_InstNum CSL_MemprotObj::modNum

This is the instance of module number i.e. L2/L1D/L1P/CONFIG

CSL_MemprotRegsOvly CSL_MemprotObj::regs

This is a pointer to the memory protection registers of the module for which memory protection is requested.

23.3.2 CSL_MemprotContext

Detailed Description

Module specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_MemprotContext::contextInfo

Context information of Memory Protection. The declaration is just a placeholder for future implementation.

23.3.3 CSL_MemprotHwSetup

Detailed Description

This is the setup structure used with the HwSetup API.

Field Documentation

Uint32* CSL_MemprotHwSetup::key

This should point to an array of 2 32 bit elements (constituting the key)

Uint16* CSL_MemprotHwSetup::memPageAttr

This should point to a table of memory page attributes

Uint16 CSL_MemprotHwSetup::numPages

This is the number of pages which need to be programmed starting from 0

23.3.4 CSL_MemprotBaseAddress

Detailed Description

This will have the base-address information for the module instance.

Field Documentation

CSL_MemprotRegsOvly CSL_MemprotBaseAddress::regs
Base-address of the memory protection registers

23.3.5 CSL_MemprotFaultStatus

Detailed Description
This will be used to query the memory fault status.

Field Documentation

Uint32 CSL_MemprotFaultStatus::addr
Memory Protection Fault Address

CSL_BitMask16 CSL_MemprotFaultStatus::errorMask
Bit Mask of the Errors

Uint16 CSL_MemprotFaultStatus::fid
Faulted ID

23.3.6 CSL_MemprotPageAttr

Detailed Description
This will be used to set/query the memory page attributes.

Field Documentation

CSL_BitMask16 CSL_MemprotPageAttr::attr
Memory Protection Page attributes

Uint16 CSL_MemprotPageAttr::page
Memory Protection Page number

23.3.7 CSL_MemprotParam

Detailed Description
This is module specific parameter. Present implementation of Memprot CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_MemprotParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation. Passed as an argument to CSL_memprotOpen().

23.4 Enumerations

This section lists the enumerations available in the MEMPROT module.

23.4.1 CSL_MemprotHwStatusQuery

enum CSL_MemprotHwStatusQuery
Enumeration for queries passed to *CSL_memprotGetHwStatus()*.
This is used to get the status of different operations or he current register settings.

Enumeration values:

CSL_MEMPROT_QUERY_FAULT	Gets the fault status from the unit. **Parameters:** *(CSL_MemprotFaultStatus* *)
CSL_MEMPROT_QUERY_PAGEATTR	Get the memory protection page attributes. **Parameters:** *(CSL_MemprotPageAttr* *)
CSL_MEMPROT_QUERY_LOCKSTAT	Memory protection Lock status. **Parameters:** *(CSL_MemprotLockStatus* *)

23.4.2 CSL_MemprotHwControlCmd

enum CSL_MemprotHwControlCmd
Enumeration for commands passed to *CSL_memprotHwControl()*.
This is used to select the commands to control the operations in the Module.

Enumeration values:

CSL_MEMPROT_CMD_LOCK	Locks the Memory Protection Unit (command argument **Parameters:** *Uint32** (An array of 2 32 bits elements constituting the key))
CSL_MEMPROT_CMD_UNLOCK	Unlocks the Memory Protection Unit (command argument **Parameters:** *Uint32** (An array of 2 32 bits elements constituting the key))
CSL_MEMPROT_CMD_PAGEATTR	Sets the page attributes **Parameters:** *(CSL_MemprotPageAttr*)*

23.4.3 CSL_MemprotLockStatus

enum CSL_MemprotLockStatus
Enumeration for queried lock status.

Enumeration values:

CSL_MEMPROT_LOCKSTAT_LOCK	Non secure Lock
CSL_MEMPROT_LOCKSTAT_UNLOCK	Non secure UnLock

23.5 Macros

#define CSL_MEMPROT_MEMACCESS_AID0 0x0400
Allowed ID '0'

#define CSL_MEMPROT_MEMACCESS_AID1 0x0800
Allowed ID '1'

#define CSL_MEMPROT_MEMACCESS_AID2 0x1000
Allowed ID '2'

#define CSL_MEMPROT_MEMACCESS_AID3 0x2000
Allowed ID '3'

#define CSL_MEMPROT_MEMACCESS_AID4 0x4000
Allowed ID '4'

#define CSL_MEMPROT_MEMACCESS_AID5 0x8000
Allowed ID '5'

#define CSL_MEMPROT_MEMACCESS_EXT 0x0200
External Allowed ID. VBus requests with PrivID >= '6' are permitted if access type is allowed

#define CSL_MEMPROT_MEMACCESS_LOCAL 0x0100
Local Access

#define CSL_MEMPROT_MEMACCESS_SR 0x0020
Supervisor Read permission

#define CSL_MEMPROT_MEMACCESS_SW 0x0010
Supervisor Write permission

#define CSL_MEMPROT_MEMACCESS_SX 0x0008
Supervisor Execute permission

#define CSL_MEMPROT_MEMACCESS_UR 0x0004
User Read permission

#define CSL_MEMPROT_MEMACCESS_UW 0x0002
User Write permission

#define CSL_MEMPROT_MEMACCESS_UX 0x0001
User Execute permission

23.6 Typedefs

typedef volatile CSL_Memprotl2RegsOvly CSL_MemprotRegsOvly
Pointer to the L2 memeory protection overlay registers

typedef CSL_MemprotObj * CSL_MemprotHandle
MEMPROT handle.

typedef void CSL_MemprotConfig
Dummy structure.

Chapter 24
PWRDWN Module

Topics

24.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within PWRDWN module.

The power-down controller allows software-driven power-down management for all of the C64x+ megamodule components. The CPU can power-down part or all of the C64x+ megamodule through the power-down controller based on its own execution thread or in response to an external stimulus from a host or global controller.These power-down features can be used to design systems for lower overall system power requirements.

24.2 Functions

This section lists the functions available in the PWRDWN module.

24.2.1 CSL_pwrdwnInit

CSL_Status CSL_pwrdwnInit (CSL_PwrdwnContext * *pContext*)

Description
CSL_pwrdwnInit(..) initializes the PWRDWN module. The function must be called before calling any other API from this CSL. This function does not modify any registers or check status. It returns status CSL_SOK. It has been kept for future use.

Arguments

```
pContext    Pointer to module-context. As PWRDWN doesn't
            have any context based information user is expected
            to pass NULL.
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
...
if (CSL_SOK != CSL_pwrdwnInit(NULL)) {
   return;
}
...
```

24.2.2 CSL_pwrdwnOpen

CSL_PwrdwnHandle CSL_pwrdwnOpen (CSL_PwrdwnObj * *pPwrdwnObj*, CSL_InstNum *pwrdwnNum*, CSL_PwrdwnParam * *pPwrdwnParam*, CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the PWRDWN instance and returns a handle to the instance. The open call sets up the data structures for the particular instance of PWRDWN device. The device can be re-opened anytime after it has been normally closed, if so required. The handle returned by this call is input as an essential argument for rest of the APIs described for this module.

Arguments

pPwrdwnObj	`Pointer to PWRDWN object.`
`pwrdwnNum`	`Instance of pwrdwn CSL to be opened.`
`pPwrdwnParam`	`Module specific parameters`
`pStatus`	`Status of the function call`

Return Value
`CSL_pwrdwnHandle`

- Valid pwrdwn handle will be returned if status value is equal to CSL_SOK.

Pre Condition
CSL_pwrdwnInit() must be called prior to this call.

Post Condition
1. The status is returned in the status variable. If status returned is

- `CSL_SOK` - Valid pwrdwn handle is returned
- `CSL_ESYS_FAIL` - The pwrdwn instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid Parameter

2. Pwrdwn object structure is populated.

Modifies
1. The status variable
2. pwrdwn object structure

Example
```
CSL_PwrdwnObj    pwrObj;
CSL_PwrdwnHandle hPwr;
CSL_Status       status;

// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);

// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Config structure
...
```

24.2.3 CSL_pwrdwnClose

CSL_Status CSL_pwrdwnClose **(** [CSL_PwrdwnHandle](#) ***hPwrdwn*** **)**

Description
This function closes the specified instance of pwrdwn.

Arguments

```
hPwrdwn        Handle to the PWRDWN instance
```

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call.

Post Condition
The PWRDWN CSL APIs can not be called until the PWRDWN CSL is reopened again using CSL_pwrdwnOpen()

Modifies
CSL_pwrdwnObj structure values

Example:

```
CSL_PwrdwnObj    pwrObj;
CSL_PwrdwnHandle hPwr;
CSL_Status       status;

// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
exit (0);

// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments fof the Config structure
...
// Close
status = CSL_pwrdwnClose(hPwr);
...
```

24.2.4 CSL_pwrdwnHwSetup

CSL_Status CSL_pwrdwnHwSetup **(** [CSL_PwrdwnHandle](#) ***hPwrdwn*, **
[CSL_PwrdwnHwSetup](#) ***** ***setup***

)

Description
It configures the PWRDWN instance registers as per the values passed in the hardware setup structure.

Arguments

```
hPwrdwn         Handle to the pwrdwn instance

setup           Pointer to hardware setup structure
```

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Hardware structure is not properly initialized

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call.

Post Condition
PWRDWN registers instance will be setup according to value passed.

Modifies
PWRDWN hardware registers

Example:

```
CSL_PwrdwnObj     pwrObj;
CSL_PwrdwnHwSetup pwrSetup;
CSL_PwrdwnHandle  hPwr;
CSL_Status        status;

// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);
// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Setup structure
...

// Setup
status = CSL_pwrdwnHwSetup(hPwr,&pwrSetup);

// Close handle
CSL_pwrdwnClose(hPwr);
...
```

24.2.5 CSL_pwrdwnGetHwSetup

CSL_Status CSL_pwrdwnGetHwSetup **(** [CSL_PwrdwnHandle](#) ***hPwrdwn*,**
[CSL_PwrdwnHwSetup](#) ***** ***setup***
)

Description
It retrieves the hardware setup parameters.

Arguments

hPwrdwn	Handle to the PWRDWN instance
setup	Pointer to hardware setup structure

Return Value
`CSL_Status`

- `CSL_SOK` - Hardware setup retrieved
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - The param passed is invalid

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call.

Post Condition
The hardware set up structure will be populated with values from the registers.

Modifies
Second parameter "setup"

Example
```
CSL_PwrdwnObj      pwrObj;
CSL_PwrdwnHwSetup pwrSetup, querySetup;
CSL_PwrdwnHandle  hPwr;
CSL_Status        status;
// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);
// Opening a handle for the Module
    hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Setup structure
...

// Setup
  CSL_pwrdwnHwSetup(hPwr,&pwrSetup);

// Query Setup
status = CSL_pwrdwnGetHwSetup(hPwr,&querySetup);

// Close handle
```

```
CSL_pwrdwnClose(hPwr);
...
```

24.2.6 CSL_pwrdwnGetHwStatus

CSL_Status CSL_pwrdwnGetHwStatus	**(**	CSL_PwrdwnHandle	***hPwrdwn*,**
		CSL_PwrdwnHwStatusQuery	***query*,**
		void *	***response***
	)		

Description
This function is used to get the value of various parameters of the PWRDWN instance. The value returned depends on the query passed.

Arguments

hPwrdwn	Handle to the PWRDWN instance
query	Query to be performed
response	Pointer to buffer to return the data requested by the query passed

Return Value
`CSL_Status`

- `CSL_SOK` - Successful completion of the query
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVQUERY` - Query command not supported
- `CSL_ESYS_INVPARAMS` - Invalid Parameters.

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call.

Post Condition
Data requested by the query is returned through the variable "response".

Modifies
The input argument "response" is modified.

Example:

```
CSL_PwrdwnObj           pwrObj;
CSL_PwrdwnHwSetup       pwrSetup;
CSL_PwrdwnHandle        hPwr;
CSL_Status              status;
CSL_PwrdwnPortData      pageSleep;

pageSleep.portNum = 0x0;

// Init Module
...
```

```
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);
// Opening a handle for the Module
  hPwr = CSL_pwrdwnOpen (&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Setup structure
...

// Setup
  CSL_pwrdwnHwSetup(hPwr,&pwrSetup);

// Hw Status Query
status = CSL_pwrdwnGetHwStatus( hPwr,
                                CSL_PWRDWN_QUERY_PAGE0_STATUS,
                                &pageSleep
                              );

// Close handle
 CSL_pwrdwnClose(hPwr);
...
```

24.2.7 CSL_pwrdwnHwSetupRaw

CSL_Status CSL_pwrdwnHwSetupRaw **(** CSL_PwrdwnHandle ***hPwrdwn*,**
CSL_PwrdwnConfig ***** ***config***
)

Description
This function initializes the device registers with the register-values provided through the config data structure. This configures registers based on a structure of register values, as compared to HwSetup, which configures registers based on structure of bit field values

Arguments

`hPwrdwn`	`Pointer to the object that holds reference to the instance of PWRDWN requested after the call`
`config`	`Pointer to the config structure containing the device register values`

Return Value
`CSL_Status`

- `CSL_SOK` - Configuration successful
- `CSL_ESYS_BADHANDLE` - Invalid handle
- `CSL_ESYS_INVPARAMS` - Configuration structure pointer is not properly initialized

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call.

Post Condition
The registers of the specified PWRDWN instance will be setup according to the values passed through the config structure.

Modifies
Hardware registers of the specified PWRDWN instance

Example

```
CSL_PwrdwnObj       pwrObj;
CSL_Status          status;
CSL_PwrdwnConfig    pwrConfig;
CSL_PwrdwnHandle    hPwr;
// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);
// Opening a handle for the Module
  hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Config structure
...

// Setup
  status = CSL_pwrdwnHwSetupRaw(hPwr,&pwrConfig);

// Close handle
CSL_pwrdwnClose(hPwr);
...
```

24.2.8 CSL_pwrdwnHwControl

CSL_Status CSL_pwrdwnHwControl	**(**	**CSL_PwrdwnHandle**	***hPwrdwn*,**
		CSL_PwrdwnHwControlCmd	***cmd*,**
		void *	***arg***
	)		

Description
This function performs various control operations on the PWRDWN instance based on the command passed.

Arguments

```
hPwrdwn          Handle to the PWRDWN instance

cmd              Operation to be performed on the PWRDWN

arg              Argument specific to the command
```

Return Value
`CSL_Status`

- `CSL_SOK` - Command execution successful
- `CSL_ESYS_INVCMD` - Invalid command
- `CSL_ESYS_BADHANDLE` - Invalid handle

- `CSL_ESYS_INVPARAMS` - Invalid Parameter

Pre Condition
CSL_pwrdwnInit(), CSL_pwrdwnOpen() must be opened prior to this call

Post Condition
Registers of the PWRDWN instance are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
Registers determined by the command

Example

```
CSL_PwrdwnObj          pwrObj;
CSL_PwrdwnHwSetup      pwrSetup;
CSL_PwrdwnHandle       hPwr;
CSL_PwrdwnPortData     pageSleep;
CSL_Status             status;

// Init Module
...
if (CSL_pwrdwnInit(NULL) != CSL_SOK)
    exit (0);
// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Setup structure
...

  // Setup
  CSL_pwrdwnHwSetup(hPwr,&pwrSetup);

// Hw Control
pageSleep.portNum = 0x1;
pageSleep.data = 0x0;

status = CSL_pwrdwnHwControl(hPwr,
                             CSL_PWRDWN_CMD_PAGE0_SLEEP,
                             &pageSleep
                             );

// Close handle
CSL_pwrdwnClose(hPwr);
...
```

24.2.9 CSL_pwrdwnGetBaseAddress

CSL_Status CSL_pwrdwnGetBaseAddress (	**CSL_InstNum**	***pwrdwnNum*,**
	CSL_PwrdwnParam *	***pPwrdwnParam*,**
	CSL_PwrdwnBaseAddress *	***pBaseAddress***
)		

Description
This function gets the base address of the given pwrdwn instance.

Arguments

`pwrdwnNum`	`Specifies the instance of the pwrdwn to be opened.`
`pPwrdwnParam`	`pwrdwn module specific parameters.`
`pBaseAddress`	`Pointer to base address structure containing base address details.`

Return Value
`CSL_Status`

- `CSL_SOK` - Successfull on getting the base address of PWRDWN.
- `CSL_ESYS_FAIL` - pwrdwn instance is not available.
- `CSL_ESYS_INVPARAMS` - Invalid parameter.

Pre Condition
None

Post Condition
Base address structure is populated.

Modifies
1. The status variable
2. Base address structure is modified.

Example

```
CSL_Status       status;
CSL_PwrdwnBaseAddress  baseAddress;
...
status = CSL_pwrdwnGetBaseAddress(CSL_PWRDWN, NULL,
                                    &baseAddress) ;
...
```

24.3 Data Structures

This section lists the data structures available in the PWRDWN module.

24.3.1 CSL_PwrdwnObj

Detailed Description
This object contains the reference to the instance of PWRDWN opened using the CSL_pwrdwnOpen().

Field Documentation

CSL_InstNum CSL_PwrdwnObj::instNum
This is the instance of PWRDWN being referred to by this object

CSL_L2pwrdwnRegsOvly CSL_PwrdwnObj::l2pwrdwnRegs
This is a pointer to the registers of the instance of L2 PWRDWN referred to by this object

CSL_PdcRegsOvly CSL_PwrdwnObj::pdcRegs
This is a pointer to the registers of the instance of PDC referred to by this object

24.3.2 CSL_PwrdwnConfig

Detailed Description
The config-structure.Used to configure the PWRDWN using CSL_pwrdwnHwSetupRaw(..).This is a structure of register values, rather than a structure of register field values like CSL_PwrdwnHwSetup

Field Documentation

Uint32 CSL_PwrdwnConfig::L2PDSLEEP0
Per page manual sleep for port0

Uint32 CSL_PwrdwnConfig::L2PDSLEEP1
Per page manual sleep for port1

Uint32 CSL_PwrdwnConfig::L2PDWAKE0
Per page manual awake for port0

Uint32 CSL_PwrdwnConfig::L2PDWAKE1
Per page manual awake for port1

Uint32 CSL_PwrdwnConfig::PDCCMD
Power down command register

24.3.3 CSL_PwrdwnContext

Detailed Description
Module specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_PwrdwnContext::contextInfo
Context information of PWRDWN. The declaration is just a placeholder for future implementation. This is a Dummy.

24.3.4 CSL_PwrdwnHwSetup

Detailed Description
This has all the fields required to configure PWRDWN at Power Up (After a Hardware Reset) or a Soft Reset. This structure is used to setup or obtain existing setup of PWRDWN using *CSL_pwrdwnHwSetup()* and *CSL_pwrdwnGetHwSetup()* functions respectively.

Field Documentation

Bool CSL_PwrdwnHwSetup::idlePwrdwn
Idle powerdown

CSL_PwrdwnL2Manual* CSL_PwrdwnHwSetup::manualPwrdwn
Manual power down setup

24.3.5 CSL_PwrdwnParam

Detailed Description
Module specific parameters. None in this implementation.

Field Documentation

void* CSL_PwrdwnParam::futureUse
Perhaps useful for future use

24.3.6 CSL_PwrdwnBaseAddress

Detailed Description
This will have the base-address information for the module instance.

Field Documentation

CSL_L2pwrdwnRegsOvly CSL_PwrdwnBaseAddress::l2pwrdwnRegs
Base-address of the L2 Powerdown registers

CSL_PdcRegsOvly CSL_PwrdwnBaseAddress::regs
Base-address of the PDC registers

24.3.7 CSL_PwrdwnPortData

Detailed Description
This will have the port specific information. It contains port number and data used in CSL_pwrdwnGetHwStatus() and CSL_pwrdwnHwControl()

Field Documentation

Bool CSL_PwrdwnPortData::portNum
Port number

CSL_BitMask8 CSL_PwrdwnPortData::data
8-bit mask

24.3.8 CSL_PwrdwnL2Manual

Detailed Description
The manual powerdown setup structure.

Field Documentation

CSL_BitMask8 CSL_PwrdwnL2Manual::port0PageSleep
Bitmask of the pages that need to be put to sleep on UMAP0

CSL_BitMask8 CSL_PwrdwnL2Manual::port0PageWake
Bitmask of the pages that need to be woken on UMAP0

CSL_BitMask8 CSL_PwrdwnL2Manual::port1PageSleep
Bitmask of the pages that need to be put to sleep on UMAP1

CSL_BitMask8 CSL_PwrdwnL2Manual::port1PageWake
Bitmask of the pages that need to be woken on UMAP1

24.4 Enumerations

This section lists the enumerations available in the PWRDWN module.

24.4.1 CSL_PwrdwnHwStatusQuery

enum CSL_PwrdwnHwStatusQuery
Default values for the config-structure Enumeration for queries passed to *CSL_pwrdwnGetHwStatus()*. This is used to get the status of different operations or to get the existing setup of PWRDWN.

Enumeration values:

CSL_PWRDWN_QUERY_PAGE0_STATUS	Gets the page0 sleep status **Parameters:** *(CSL_PwrdwnPortData* *)
CSL_PWRDWN_QUERY_PAGE1_STATUS	Gets the page1 sleep status **Parameters:** *(CSL_PwrdwnPortData* *)

24.4.2 CSL_PwrdwnHwControlCmd

enum CSL_PwrdwnHwControlCmd
Enumeration for queries passed to *CSL_pwrdwnHwControl()*.
This is used to select the commands to control the operations existing setup of PWRDWN. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values:

CSL_PWRDWN_CMD_PAGE0_SLEEP	Manual power down, port0 or port1, page0 sleep **Parameters:** *(CSL_PwrdwnPortData* *)
CSL_PWRDWN_CMD_PAGE1_SLEEP	Manual power down, port0 or port1, page1 sleep **Parameters:** *(CSL_PwrdwnPortData* *)
CSL_PWRDWN_CMD_PAGE0_WAKE	Manual power down, port0 or port1, page0 wake **Parameters:** *(CSL_PwrdwnPortData* *)
CSL_PWRDWN_CMD_PAGE1_WAKE	Manual power down, port0 or port1, page1 wake **Parameters:** *(CSL_PwrdwnPortData* *)

24.5 Typedefs

typedef CSL_PwrdwnObj * CSL_PwrdwnHandle
Pointer to the powerdown object. This handle contains the reference to the instance of PWRDWN opened CSL_pwrdwnOpen().

Chapter 25
TSC Module

Topics

25.1 Overview

This chapter describes the Functions within TSC module.

Time Stamp Counter is a free running 64-bit CPU counter that advances each CPU clock after counting is enabled. The counter is accessed using two 32-bit read-only control registers, Time Stamp Counter Registers – Low (TSCL) and Time Stamp Counter Registers – High (TSCH). The counter is enabled by writing to TSCL. The value written is ignored. Once enabled, counting cannot be disabled under program control. Counting is disabled in the following cases:

- After exiting the reset state.
- When the CPU is fully powered down.

25.2 Functions

This section lists the functions available in the TSC module.

25.2.1 CSL_tscEnable

void CSL_tscEnable (void)

Description
This API enables the 64 bit time stamp counter. Time Stamp Counter stops only upon Reset or Power down. When time stamp counter is enabled (following a reset or power down of the CPU) it will initialize to 0 and starts incrementing once per CPU cycle. The reset time stamp counter operation is not allowed.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
Time Stamp Counter value starts incrementing.

Modifies
None

Example

```
...
CSL_tscEnable();
...
```

25.2.2 CSL_tscRead

CSL_Uint64 CSL_tscRead (void)

Description
Reads the 64-bit Time Stamp Counter and returns the 64 bit counter value.

Arguments
None

Return Value
`CSL_Uint64`

- 64 Bit Time Stamp Counter Value

Pre Condition
The Time Stamp Counter must be succesfully enabled via *CSL_tscEnable ()* before calling this function.

Post Condition
None

Modifies
None

Example

```
CSL_Uint64      counterVal;
...
CSL_tscEnable();
counterVal  = CSL_tscRead ();
...
```