TMS320C6472 Chip Support Library

API Reference Guide

Publication Date: October 2010

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.

Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
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
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	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 TMS320C6472 CSL.

The purpose of this document is to identify the set of published Chip Support Library (CSL) APIs for the TMS320C6472 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
BWMNGMT	Bandwidth Management
CFG	Configuration
CSL	Chip Support Library
DAT	Data module
DDR	Double Data Rate
DTF	DSP Trace Formatter
ECTL	Ethernet Multi core interrupt Control
EDMA	Enhanced Direct Memory Access
EDC	Error Detection and Correction
EMAC	Ethernet Media Access Controller
ETB	Embedded Trace Buffer
GPIO	General Purpose Input/Output
HPI	Host Port Interface
I2C	Inter Integrated Circuit
IDMA	Internal DMA
INTC	Interrupt Controller
MDIO	Management Data Input /Output
MEMPROT	Memory Protection
PDMA	Peripheral Direct Memory Access
PIM	PDMA Interface Manager
PLLC	PLL Controller
PSC	Power and Sleep Controller

PWRDWN	Power Down
SRIO	Serial Rapid IO
SMC	Shared Memory Controller
SMCP	Shared Memory Controller Profiler
TSC	Time Stamp Counter
TSIP	Telecom Serial Interface Port
UTOPIA	Universal Test and Operations Interface for ATM

TABLE OF CONTENTS

Chapter 1
Introduction

Topics

1.1 Introduction

The Chip Support Library constitutes a set of well-defined APIs that abstract low-level details of the underlying SoC device so that a 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 the OS. This helps in improving portability of code written using the CSL.

1.2 Overview

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

It is important to note that the 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.

The CSL does not model the device state machine. However, should there exist a mandatory (hardware dictated) sequence (possibly automatically executed) of register reads/writes to setup the device in chosen “operating modes” as per the device datasheet, then the 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 the CSL scalable to fit the specific usage scenarios and ease the effort to ROM a CSL based application.

1.3 CSL Interface

The 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, the 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 as header files.

1.4 Functional Layer

The CSL Functional Layer for the TMS320C6472 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.

Interface functions exported by this layer are "run to completion," meaning that they shall not support asynchronous behavior or deferred completion. If the peripheral hardware has the 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 the 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_BitMask16	**Uint16**
CSL_BitMask32	**Uint32**
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, the 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_U**art**H**andle

Table 4: CSL 2.x naming conventions

Format	Namespace	Type
<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.	MCBSP_RCV_START Here MCBSP is <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 the 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 16-bit signed binary numbers, that allows us to represent 32K 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.
CSL_ESYS_ALREADY_INITIALIZED	-10	Module already initialized

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	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_DMA, 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, the 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 0x01940000 and the other at address 0x01944000, the register overlay for such a device is specified as follows:

```
typedef struct {
  Uint32 CTL; /* Timer Control Register */
  Uint32 PRD; /* Timer Period Register */
  Uint32 CNT; /* Timer Counter Register */
} CSL_TimerRegs;
typedef volatile CSL_TimerReg *CSL_TimerRegsOvly
```

CSL_TIMER_0_REGS	**CTL [+0x00]**
(0x01940000)	**PRD [+0x04]**
	CNT [+0x08]
CSL_TIMER_1_REGS	**CTL [+0x00]**
(0x01944000)	**PRD [+0x04]**
	CNT [+0x08]

Figure 1: Register Layer Overlay Structure

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 buses. Typically, the processor hardware wiring will be such that the CPU always gets to read/write its memory mapped peripherals in "Native Endian" format, i.e., ready for CPU interpretation. Should there be an Endian mismatch between the 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 this 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.

The following section describes the CSL INTC functionality for C6472.

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 the 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.

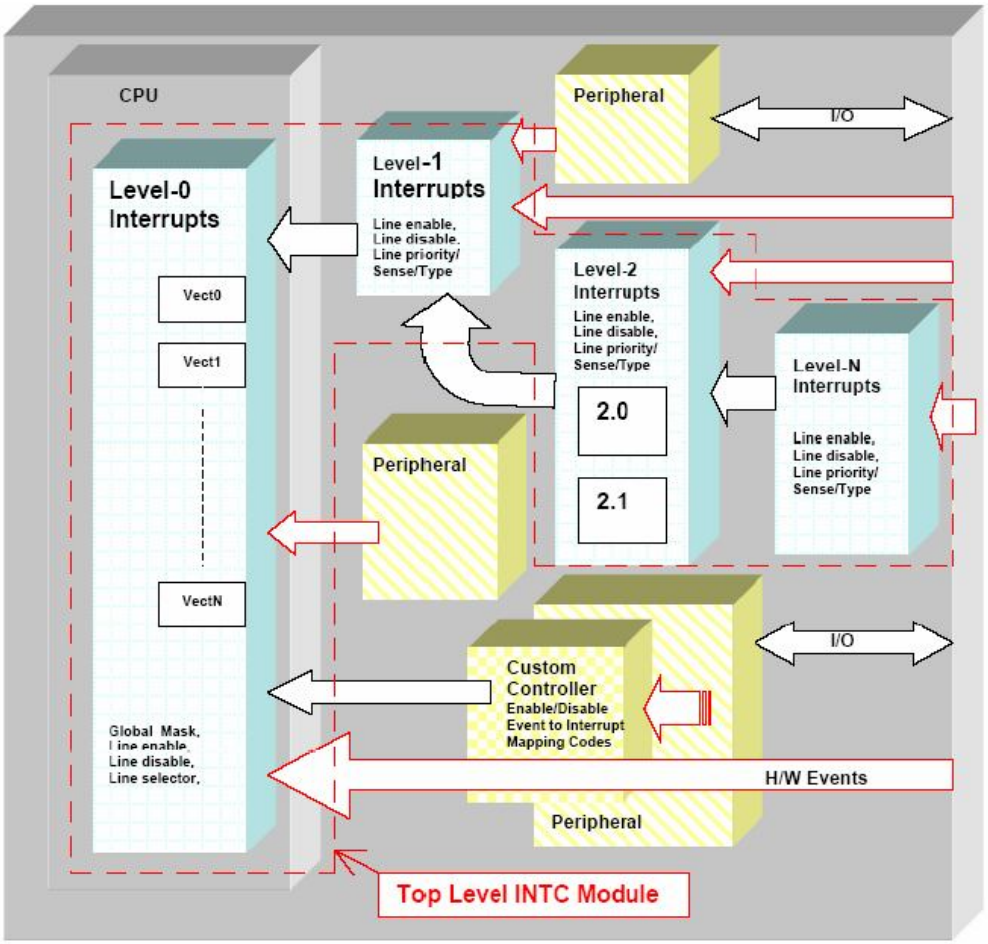

Figure 2 INTC Controller block diagram

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 through INTCn).

During the CSL initialization phase, the INTC module initialization function CSL_intcInit() is called. This resets 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 of 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 the INTC module (via CSL_intcInit()), the user can choose to initialize 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 an arbitrary data object (void*) to be passed to the 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 an 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 DMA peripheral as for devices that have the EDMA peripheral.

2.2 Functions

This section lists the functions available in the DAT module.

2.2.1 DAT_open

Int16 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_SOK

Pre Condition
None

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

Modifies
EDMA registers

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);
```

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];
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_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];
```

```
Uint8       fillVal;

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;
DAT_fill(&dst,256,(Uint32*)&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];
Uint32      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];
Uint32      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];
Uint32      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;

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.

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

- CSL_SOK - Valid DDR2 instance handle will be returned.
- CSL_ESYS_FAIL - DDR2 instance is invalid.

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 is CSL_SOK then a valid DDR2 handle is returned. If status is NULL then the DDR2 instance 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 - external memory interface 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_Status         status;
CSL_Ddr2Obj        ddr2Obj;
CSL_Ddr2Handle   hDdr2;

if (CSL_SOK != CSL_ddr2Init(NULL))
{
      return;
}
hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &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_Ddr2Obj       ddr2Obj;
CSL_Ddr2Timing1 tim1 = CSL_DDR2_TIMING1_DEFAULTS;
CSL_Ddr2Timing2 tim2 = CSL_DDR2_TIMING2_DEFAULTS;
CSL_Ddr2Settings set = CSL_DDR2_SETTING_DEFAULTS;
CSL_Ddr2HwSetup hwSetup;

if (CSL_SOK != CSL_ddr2Init(NULL))
{
      return;
}
hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &status);
hwSetup.refreshRate = (Uint16)0x753;
hwSetup.timing1Param = &tim1;
hwSetup.timing2Param = &tim2;
hwSetup.setParam = &set;
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 - 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_Ddr2Obj ddr2Obj;
CSL_Status status;
CSL_Ddr2Timing1 tim1;
CSL_Ddr2Timing2 tim2;
CSL_Ddr2Settings set;
CSL_Ddr2HwSetup hwSetup;

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

hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &status);
...

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_Ddr2Obj       ddr2Obj;

 CSL_Ddr2SelfRefresh command;
 if (CSL_SOK != CSL_ddr2Init(NULL))
{
      return;
}
hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &status);
...
command = CSL_DDR2_SELF_REFRESH_DISABLE;
...
status = CSL_ddr2HwControl(   hDdr2,
                              CSL_DDR2_CMD_SELF_REFRESH,
                              &command);
```

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 value present 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

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;
    CSL_Ddr2Obj       ddr2Obj;

    if (CSL_SOK != CSL_ddr2Init(NULL))
    {
        return;
    }
    hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &status);
    ...
    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;
CSL_Ddr2Obj       ddr2Obj;

if (CSL_SOK != CSL_ddr2Init(NULL))
{
      return;
}
hDdr2 = CSL_ddr2Open(&ddr2Obj, CSL_DDR2, NULL, &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 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 referred 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::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.

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

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 (response type: (CSL_Ddr2ModIdRev*))
CSL_DDR2_QUERY_REFRESH_RATE	Get the EMIF refresh rate information (response type: *Uint16* *)
CSL_DDR2_QUERY_SELF_REFRESH	Get self refresh bit value (response type: (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. (Response type: *Uint8**)
CSL_DDR2_QUERY_ENDIAN	Gets the current endian of DDR2 emif from the SDRAM Status register. (response type: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: argument (CSL_Ddr2SelfRefresh *)
CSL_DDR2_CMD_REFRESH_RATE	Enters the Refresh rate value: argument (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. Argument (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

3.5 Macros

#define CSL_DDR2_CONFIG_DEFAULTS

Value:

```
{ \
    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

Value:

```
{  \
    (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\
}
```

The default values of DDR2 SDRAM settings.

#define CSL_DDR2_TIMING1_DEFAULTS

Value:

```
{\
    (Uint16)CSL_DDR2_TIMING1_TRFC_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TRP_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TRCD_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TWR_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TRAS_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TRC_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TRRD_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING1_TWTR_DEFAULT \
}
```

The default values of DDR2 SDRAM Timing1 Control structure.

#define CSL_DDR2_TIMING2_DEFAULTS

Value:

```
{ \
    (Uint8)CSL_DDR2_TIMING2_T_ODT_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING2_TSXNR_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING2_TSXRD_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING2_TRTP_DEFAULT, \
    (Uint16)CSL_DDR2_TIMING2_TCKE_DEFAULT \
}
```

The default values of DDR2 SDRAM Timing2 Control structure.

Chapter 4
DTF Module

Topics

4.1 Overview

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

GEM's trace port sends the trace data to DTF (DSP Trace Formatter). This trace data is 22bits per clock cycle in FullGEM. The ETB has a 32bit data interface. The primary function of DTF is to pack the 22bit trace data into a 32bit packet. All the DTF's MMR's are located in the chip level register space and thus it does not have a VBUS interface. DTF also forwards the trace data as is to the SPM (Static Pin Merge).

4.2 Functions

4.2.1 CSL_dtfInit

CSL_Status CSL_dtfInit (CSL_DtfContext * *pContext*)

Description
This is the initialization function for the DTF. This function must be called before calling any other API from this CSL. This function is idem-potent. Currently, the function just returns status CSL_SOK, without doing anything.

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_dtfInit(NULL);
...
```

4.2.2 CSL_dtfOpen

CSL_DtfHandle CSL_dtfOpen (CSL_DtfObj * *pDtfObj*, CSL_InstNum *dtfNum*, CSL_DtfParam * *pDtfParam*, CSL_Status * *pStatus*)

Description
This function returns the handle to the DTF controller instance. This handle is passed to all other CSL APIs.

Arguments

pDtfObj	Pointer to dtf object.
dtfNum	Instance of DSP DTF to be opened. There are six instances of the dtf available.
pDtfParam	Module specific parameters.

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

Return Value
`CSL_DtfHandle`

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

Pre Condition
CSL_dtfInit() must be called successfully in order before calling **CSL_dtfOpen()**.

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

   - `CSL_SOK` - Valid dtf handle is returned
   - `CSL_ESYS_FAIL` - The dtf instance is invalid
   - `CSL_ESYS_INVPARAMS` - Invalid parameter

2. DTF object structure is populated

Modifies
1. The status variable
2. DTF object structure

Example

```
CSL_Status         status;
CSL_DtfObj         dtfObj;
CSL_DtfHandle      hDtf;
...
hDtf = CSL_dtfOpen(&dtfObj, CSL_DTF_0, NULL, &status);
...
```

4.2.3 CSL_dtfClose

CSL_Status CSL_dtfClose (CSL_DtfHandle ***hDtf*** **)**

Description
This function closes the specified instance of DTF.

Arguments

```
hDtf        Handle to the DTF
```

Return Value
`CSL_Status`

- `CSL_SOK` - Close successful
- `CSL_ESYS_BADHANDLE` - Invalid handle

Pre Condition
Both **CSL_dtfInit()** and **CSL_dtfOpen()** must be called successfully in order before calling **CSL_dtfClose()**.

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

Modifies
Obj structure values

Example

```
CSL_DtfHandle           hDtf;
CSL_Status              status;
...

status = CSL_dtfClose(hDtf);
...
```

4.2.4 CSL_dtfHwControl

CSL_Status CSL_dtfHwControl (CSL_DtfHandle ***hDtf*,**
CSL_DtfControlCmd *cmd*,
void * *arg*
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of DTF.

Arguments

hDtf	DTF handle returned by successful 'open'
cmd	The command to this API indicates the action to be taken on DTF. Control command, refer @a CSL_DtfControlCmd for the list of commands supported
arg	An optional argument. Optional argument as per the control command, void* casted

Return Value
CSL_Status

- CSL_SOK - Status info return successful.
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVCMD - Invalid command

Pre Condition
Both **CSL_dtfInit()** and **CSL_dtfOpen()** must be called successfully in order before calling **CSL_dtfHwControl()**.
Refer to *CSL_DtfHwControlCmd* for the argument type (*void**) that needs to be passed with the control command

Post Condition
None

Modifies
The hardware registers of DTF.

Example

```
CSL_Status status;
Uint32  arg;
CSL_DtfHandle hDtf;
...
// Init successfully done
...
// Open successfully done
...

arg = 1;
status = CSL_dtfHwControl(hDtf,
                          CSL_DTF_CMD_SET_DTFENABLE,
                          &arg);
...
```

4.2.5 CSL_dtfGetHwStatus

CSL_Status CSL_dtfGetHwStatus (

CSL_DtfHandle	*hDtf*,		
CSL_DtfHwStatusQuery	*query*,		
void *	*response*		
)			

Description

Gets the status of different operations or some setup-parameters of DTF. The status is returned through the third parameter.

Arguments

hDtf	DTF handle returned by successful 'open'
query	The query to this API of DTF which indicates the status to be returned. Query command, refer CSL_DtfHwStatusQuery for the list of query commands supported
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 - Status info return successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Invalid query command

Pre Condition
Both **CSL_dtfInit()** and **CSL_dtfOpen()** must be called successfully in order before calling **CSL_dtfGetHwStatus()**. Refer to *CSL_DtfHwStatusQuery* for the argument to be passed along with the corresponding query command.

Post Condition
None

Modifies
Third parameter response

Example

```
CSL_DtfHandle       hDtf;
CSL_Status          status;
Uint32              response;
...

status = CSL_dtfGetHwStatus(hDtf,
                            CSL_DTF_QUERY_DTFOWN_STATUS,
                            &response);
...
```

4.2.6 CSL_dtfGetBaseAddress

CSL_Status CSL_dtfGetBaseAddress (

CSL_InstNum	***dtfNum*,**
CSL_DtfParam *	***pDtfParam*,**
CSL_DtfBaseAddress *	***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_dtfOpen()** 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. MMR's go to an alternate location.

Arguments

dtfNum	Specifies the instance of the dtf to be opened.
pDtfParam	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 dtf
- 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_DtfBaseAddress  baseAddress;
...
status = CSL_dtfGetBaseAddress(CSL_DTF_0, NULL, &baseAddress);
...
```

4.3 Data Structures

4.3.1 CSL_DtfBaseAddress

This structure will have the base-address information for the peripheral instance.

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

Field Documentation

CSL_DtfRegsOvly CSL_DtfBaseAddress::regs
Base-address of the Configuration registers of DTF.

4.3.2 CSL_DtfContext

Detailed Description
DTF specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_DtfContext::contextInfo
Context information of DTF. The below declaration is just a place-holder for future implementation.

4.3.3 CSL_DtfObj

Detailed Description
This structure/object holds the context of the instance of DTF opened using ***CSL_dtfOpen()*** function.
Pointer to this object is passed as DTF Handle to all DTF CSL APIs. ***CSL_dtfOpen()*** function intializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_DtfObj::perNum
Instance of DTF being referred by this object

CSL_DtfRegsOvly CSL_DtfObj::regs
Pointer to the register overlay structure of the DTF

4.3.4 CSL_DtfParam

Detailed Description
DTF specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_DtfParam::flags

Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

4.4 Enumerations

4.4.1 CSL_DtfControlCmd

This is the set of control commands that are passed to ***CSL_dtfHwControl()*** , 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_DTF_CMD_ENA_AOWN_APP — Setup the DTF ownership to Application
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_ENA_AOWN_EMU — Setup the DTF ownership to Emulation
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_FLUSH_DTFFLUSH — Setup the DTF Flush
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_SET_DTFENABLE — Setting the DTFEABLE
Parameters: 0 - Disable
1 - Enable
Returns: CSL_SOK

CSL_DTF_CMD_DIS_SPMDISABLE — Disable trace output to Static-Pin_merge
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_ENA_SPMDISABLE — Enable trace output to Static-Pin_merge
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_RELEASE_OWERNERSHIP — Releasing the DTF ownership
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_CLAIM_OWERSHIP — Claiming the DTF ownership
Parameters: *None*
Returns: CSL_SOK

CSL_DTF_CMD_ENA_OWNERSHIP — Enabling the DTF ownership
Parameters: *None*
Returns: CSL_SOK

4.4.2 CSL_DtfHwStatusQuery

This is the set of query commands to get the status of various operations in DTF
The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_DTF_QUERY_AOWNERSHIP — Queries the DTF ownership
Parameters :*(Uint32 *)*
Returns: CSL_SOK

CSL_DTF_QUERY_DTFOWN_STATUS — Queries DTF ownership state control
Parameters :*(Uint32 *)*
Returns: CSL_SOK

CSL_DTF_QUERY_DTFENABLE — Queries DTFENABLE bit
Parameters :*(Uint32 *)*
Returns: CSL_SOK

CSL_DTF_QUERY_SPMDISABLE — Queries Static-Pin-Merger
Parameters :*(Uint32 *)*
Returns: CSL_SOK

4.5 Macros

#define CSL_DTF_OWNERSHIP_RELEASE 0
Value to release the DTF ownership

#define CSL_DTF_OWNERSHIP_CLAIM 1
Value to claim the DTF ownership

#define CSL_DTF_OWNERSHIP_ENABLE 2
Value to enable the DTF ownership

#define CSL_DTF_DTFENABLE_DISABLE 0
Value to disable DTFENABLE

#define CSL_DTF_DTFENABLE_ENABLE 1
Value to enable DTFENABLE

#define CSL_DTF_SPM_DISABLE 0
Value to disable the SPM

#define CSL_DTF_SPM_ENABLE 1
Value to enable the SPM

4.6 Typedefs

typedef struct CSL_DtfObj CSL_DtfObj
This structure/object holds the context of the instance of DTF opened using ***CSL_dtfOpen()*** function.

typedef CSL_DtfObj * CSL_DtfHandle
This is a pointer to **CSL_DtfObj** and is passed as the first parameter to all DTF CSL APIs.

Chapter 5
ECTL MODULE

Topics

5.1 Overview

This chapter details the APIs of the ECTL module.

The Ethernet Multicore Interrupt Control module is designed to efficiently route a single set of interrupts from EMAC and MDIO to multiple cores on a multicore device. It also incorporates interrupt pacing functionality for the Ethernet Tx and Rx pulse interrupts. This module also consolidates all the interrupt signals from the EMAC into three interrupts per core and to 6 such cores. It also provides enabling and disabling of interrupts from the EMAC to the DSP. ECTL module APIs should not be called if EMAC module APIs are used in the application. EMAC APIs configure the ECTL module accordingly.

5.2 Functions

This section lists the functions available in the ECTL module.

5.2.1 ECTL_config

Uint32 ECTL_config (ECTL_Config * pEctlConfig, int instNum)

Description
Configures the ECTL module for a specific core and channel. Call this function multiple times to configure all the cores and channels.

Arguments

pEctlConfig	pointer to config structure
instNum	module instance number(0 or 1)

Return Value
Uint32

- 0 - Success
- ECTL_ERROR_INVALID - A calling parameter is invalid

Pre Condition
None.

Post Condition
ECTL module will be configured.

Modifies
Memory mapped registers ECTL are modified.

Example

```
ECTL_Config *pEctlConfig;

pEctlConfig->coreNumber = 0;
pEctlConfig->channelNumber = 1;

pEctlConfig->rxTxIntrCntl = ECTL_COMMON_INTR_HOST
                          |ECTL_COMMON_INTR_MDIO_LINT |
                          ECTL_RECEIVE_INTR_RX1 |
                          ECTL_TRANSMIT_INTR_TX1;
pEctlConfig->rxPaceTimeDelay = 0xffff;
pEctlConfig->txPaceTimeDelay = 0xffff;
pEctlConfig->rxPaceDivCount  = 0x0f;
pEctlConfig->txPaceDivCount  = 0x0f;
pEctlConfig->prescale        = 0xff00;


ECTL_config(pEctlConfig, 0);
```

5.2.2 ECTL_getStatus

Uint32 ECTL_getStatus (ECTL_Status * pStatus, int instNum)

Description
Gets the ECTL module status for a specific core and channel. Call this function multiple times to get the status of all the cores and channels

Arguments

pStatus	pointer to status structure
instNum	module instance number(0 or 1)

Return Value
Uint32

- 0 - Success
- ECTL_ERROR_INVALID - A calling parameter is invalid

Pre Condition
None

Post Condition
ECTL status structure contains the values read from the registers

Modifies
Modifies the structure passed in as argument.

Example

```
ECTL_Status *pStatus;

pStatus->coreNumber = 0;
pStatus->channelNumber = 1;

ECTL_getStatus(pStatus, 0);
```

5.3 Data Structures

This section lists Data Structures available in the ECTL module.

5.3.1 ECTL_Config

Detailed Description
This structure is used for configuring the ECTL module.

Field Documentation

int ECTL_Config::channelNumber
Tx or Rx channel number (There are 8 channels from 0 to 7)

int ECTL_Config::coreNumber
core number (There are 6 cores from 0 to 5)

Uint16 ECTL_Config::prescale
Reload value for the prescalar configuration value

Uint8 ECTL_Config::rxPaceDivCount
Rx pacer divide by N count configuration value

Uint16 ECTL_Config::rxPaceTimeDelay
Rx pacer time delay configuration value

Uint32 ECTL_Config::rxTxIntrCntl
Common, Receive and Transmit interrupts control

Uint8 ECTL_Config::txPaceDivCount
Tx pacer divide by N count configuration value

Uint16 ECTL_Config::txPaceTimeDelay
Tx pacer time delay configuration value

5.3.2 ECTL_Status

Detailed Description
This structure is used to get the status values of ECTL.

Field Documentation

int ECTL_Status::channelNumber
Tx or Rx channel number (There are 8 channels from 0 to 7)

int ECTL_Status::coreNumber
core number (There are 6 cores from 0 to 5)

Uint16 ECTL_Status::prescale
Reload value for the prescalar configuration value

Uint8 ECTL_Status::rxDivCount
Rx pacer Divide by N count configuration value status

Uint16 ECTL_Status::rxTimeDelay
Rx pacer time delay configuration value status

Uint32 ECTL_Status::rxTxIntrCntlStatus
Common, Receive and Transmit interrupts control status

Uint8 ECTL_Status::txDivCount
Tx pacer Divide by N count configuration value status

Uint16 ECTL_Status::txTimeDelay
Tx pacer time delay configuration value status

5.4 Macros

#define ECTL_COMMON_INTR_HOST 0x00000002
Host Error interrupt

#define ECTL_COMMON_INTR_MDIO_LINT 0x00000008
Serial interface link change interrupt, Indicates change in the state of the PHY link

#define ECTL_COMMON_INTR_MDIO_USER 0x00000010
Serial interface user command event complete interrupt

#define ECTL_COMMON_INTR_STAT 0x00000004
Statistics interrupt

#define ECTL_RECEIVE_INTR_RX0 0x00010000
Channel 0 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX1 0x00020000
Channel 1 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX2 0x00040000
Channel 2 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX3 0x00080000
Channel 3 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX4 0x00100000
Channel 4 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX5 0x00200000
Channel 5 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX6 0x00400000
Channel 6 Receive Queue Interrupt

#define ECTL_RECEIVE_INTR_RX7 0x00800000
Channel 7 Receive Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX0 0x00000100
Channel 0 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX1 0x00000200
Channel 1 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX2 0x00000400
Channel 2 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX3 0x00000800
Channel 3 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX4 0x00001000
Channel 4 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX5 0x00002000
Channel 5 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX6 0x00004000
Channel 6 Transmit Queue Interrupt

#define ECTL_TRANSMIT_INTR_TX7 0x00008000
Channel 7 Transmit Queue Interrupt

#define ECTL_RXTX_DIV_SM_COUNT 0x1
Divide by N count SM is in COUNT state

#define ECTL_RXTX_DIV_SM_OUTPUT 0x2
Divide by N count SM is in OUTPUT state

#define ECTL_RXTX_DIV_SM_WATING 0x0
Divide by N count SM is in WAITING state

#define ECTL_RXTX_TIM_SM_DELAY 0x1
Timed Delay SM is in DELAY state

#define ECTL_RXTX_TIM_SM_OUTPUT 0x2
Timed Delay SM is in OUTPUT state

#define ECTL_RXTX_TIM_SM_WATING 0x0
Timed Delay SM is in WAITING state

#define ECTL_ERROR_INVALID 4
Function or calling parameter is invalid

#define ECTL0_REGS ((CSL_EctlRegs *)CSL_ECTL_0_REGS)
ECTL Module instance 0 registers

#define ECTL1_REGS ((CSL_EctlRegs *)CSL_ECTL_1_REGS)
Ectl Module instance 1 registers

#define NUM_OF_CORES 6
Number of cores

#define NUM_OF_CHANNELS 8
Number of channels

#define ECTL_rxDivCountReset (channelNumber , instNum)
CSL_FINST(ECTL##instNum _REGS ->RPCFG[channelNumber], ECTL_RPCFG_CR, RESET)
This macro resets the Divide by N count for Rx channel

#define ECTL_rxDivCountState (channelNumber , instNum)
CSL_FEXT(ECTL##instNum _REGS ->RPSTAT[channelNumber], ECTL_RPSTAT_DIV_SM)
This macro reads the state of Divide by N count State Machine for Rx channel

#define ECTL_rxTimeCounterReset (channelNumber , instNum)
CSL_FINST(ECTL##instNum _REGS ->RPCFG[channelNumber], ECTL_RPCFG_TR, RESET)
This macro resets the time delay counter for Rx channel

#define ECTL_rxTimeCounterState (channelNumber, instNum)
CSL_FEXT(ECTL##instNum _REGS ->RPSTAT[channelNumber], ECTL_RPSTAT_TIM_SM)
This macro reads the state of Timed Delay State Machine for Rx channel

#define ECTL_txDivCountReset (channelNumber , instNum)
CSL_FINST(ECTL##instNum _REGS ->TPCFG[channelNumber], ECTL_TPCFG_CR, RESET)
This macro resets the Divide by N count for Tx channel

#define ECTL_txDivCountState (channelNumber , instNum)
CSL_FEXT(ECTL##instNum _REGS ->TPSTAT[channelNumber], ECTL_TPSTAT_DIV_SM)
This macro reads the state of Divide by N count State Machine for Tx channel

#define ECTL_txTimeCounterReset (channelNumber , instNum)
CSL_FINST(ECTL##instNum _REGS ->TPCFG[channelNumber], ECTL_TPCFG_TR, RESET)
This macro resets the time delay counter for Tx channel

#define ECTL_txTimeCounterState (channelNumber, instNum)
CSL_FEXT(ECTL##instNum _REGS ->TPSTAT[channelNumber], ECTL_TPSTAT_TIM_SM)
This macro reads the state of Timed Delay State Machine for Tx channel

Chapter 6
EDC Module

Topics

6.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within EDC module.
L1P and L2 support error detection and correction mechanism to protect the program code and static data which are not frequently changed. L1p support error detectction and L2 support error detection and correction mechanism.

6.2 Functions

This section lists the functions available in the EDC module.

6.2.1 CSL_edcEnable

CSL_Status CSL_edcEnable (CSL_EdcMem *edcMem*)

Description
Enables the EDC for the specified memory.

Arguments

edcMem	Specifies what memory EDC is to be enabled

Return Value
CSL_Status

- CSL_SOK - EDC disable for specified memory is successful
- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
CSL_Status      status;
...
status = CSL_edcEnable (CSL_EDC_L1P);
...
```

6.2.2 CSL_edcDisable

CSL_Status CSL_edcDisable (CSL_EdcMem *edcMem*)

Description
Disables the EDC for the specified memory.

Arguments

edcMem	Specifies what memory EDC is to be disabled

Return Value
CSL_Status

- CSL_SOK - EDC disable for specified memory is successful
- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
CSL_Status          status;
...
status = CSL_edcDisable (CSL_EDC_L1P);
...
```

6.2.3 CSL_edcSuspend

CSL_Status CSL_edcSuspend (CSL_EdcMem *edcMem*)

Description
Suspend the EDC for the specified memory.

Arguments
`edcMem` Specifies what memory EDC is to be suspend

Return Value
`CSL_Status`

- `CSL_SOK` - EDC suspend for specified memory is successful
- `CSL_ESYS_INVPARAMS` - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
CSL_Status          status;
...
status = CSL_edcSuspend (CSL_EDC_L1P);
...
```

6.2.4 CSL_edcClear

CSL_Status CSL_edcClear (CSL_EdcMem *edcMem*, CSL_EdcClrAccessType *edcAccessType*)

Description
Clears the Address of the parity error for the specified memory along with the access type parity error bit.

Arguments

edcMem	Specifies what memory EDC error address is to be cleared
edcAccessType	Specifies what fetch type parity error bit or parity error count type is to be cleared.

Return Value
CSL_Status

- CSL_SOK - Address of parity error clear is successful
- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
EDC registers

Example

```
CSL_Status          status;
...
status = CSL_edcClear (CSL_EDC_L1P, CSL_EDC_DCLR);
...
```

6.2.5 CSL_edcGetErrorAddress

CSL_Status CSL_edcGetErrorAddress (CSL_EdcMem ***edcMem***,
CSL_EdcAddrInfo * ***edcAddr***
)

Description
Gets the Address location of the parity error.

Arguments

edcMem	Specifies what memory EDC Address Info is to be acquired for.
edcAddr	Structure for returning Address, L2 Way, SRAM/Cache info bitposition for error.

Return Value
CSL_Status

- CSL_SOK - EDC get error address is successful

- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
CSL_Status          status;
CSL_EdcAddrInfo    edcAddr
...
status = CSL_edcGetErrorAddress (CSL_EDC_L1P, &edcAddr);
...
```

6.2.6 CSL_edcGetHwStatus

CSL_Status CSL_edcGetHwStatus (CSL_EdcMem ***edcMem***,
CSL_EdcHwStatusQuery ***query***,
void * ***response***
)

Description
Gets the requested HW Status of the specified memory.

Arguments

edcMem	Specifies what memory EDC status is to be obtained.
query	The query to this API which indicates the status to be returned.
response	Placeholder to return the status.

Return Value
CSL_Status

- CSL_SOK - EDC get error address is successful
- CSL_ESYS_INVPARAMS - The parameter passed is invalid
- CSL_ESYS_INVQUERY - Query command not supported

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
    CSL_Status          status;
    Uint16              enStat;
    ...
    status = CSL_edcGetHwStatus (CSL_EDC_L1P, CSL_EDC_QUERY_ENABLESTAT,
                                 (void *)&enStat);
    ...
```

6.2.7 CSL_edcPageEnable

CSL_Status CSL_edcPageEnable (Uint32 *mask*,
CSL_EdcUmap *umap*
)

Description
Enables the pages for EDC specified by a 32-bit mask.

Arguments

mask	Specifies what pages of the given map(s) are to be enabled by setting the bit corresponding to the page to 1
umap	Specifies which map(s) to apply mask to (MAP0, MAP1, or BOTH)

Return Value
CSL_Status

- CSL_SOK - Enable pages for EDC is successful
- CSL_ESYS_INVPARAMS - The parameter passed is invalid

Pre Condition
None

Post Condition
None

Modifies
None

Example
```
    CSL_Status          status;
    ...
    status = CSL_edcPageEnable (0x1, CSL_EDC_UMAP0);
    ...
```

6.3 Data Structures

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

6.3.1 CSL_EdcAddrInfo

Detailed Description
CSL_EdcAddrInfo has all the fields required locate the parity error

Field Documentation

Uint32 CSL_EdcAddrInfo::addr
Address of the parity error - 5 LSBs always 0

CSL_EdcAddrL2way **CSL_EdcAddrInfo::l2way**
The cache way the error was detected in if in cache

CSL_EdcAddrSram **CSL_EdcAddrInfo::sram**
Parity error was detected in SRAM or Cache

Uint8 CSL_EdcAddrInfo::bitPos
Bit Position of the sigle bit error in the 32 bit word (of the 256 bit block address returned in the ADDR field of L2EDADDR register) identified by the BITPOS[4:0] (word32Bit) of L2EDSTAT register

Uint8 CSL_EdcAddrInfo::word32Bit
32 bit word location of the 256 bit block address returned in the ADDR field of L2EDADDR register

6.3.2 CSL_EdcStatusInfo

Detailed Description
CSL_EdcStatusInfo used to get the error status for L1 and L2

Field Documentation

Uint8CSL_EdcStatusInfo::enStat
Enable status for L1 and L2

Uint8CSL_EdcStatusInfo::disStat
Disable status for L1 and L2

Uint8CSL_EdcStatusInfo::suspStat
Suspend status for L1 and L2

Uint8CSL_EdcStatusInfo::prgErr
Intstruction fetch errror status for L1 and L2

Uint8CSL_EdcStatusInfo::dmaErr

DMA errror status for L1 and L2

Uint8CSL_EdcStatusInfo::dataErr
Data fetch errror status for L2

6.4 Enumerations

This section lists the enumerations available in the EDC module.

6.4.1 CSL_EdcMem

enum CSL_EdcMem
Memory Specifier for EDC. Used to indicate which memories (L1P or L2) are to be affected by the API.

Enumeration values:

CSL_EDC_L1P	L1P's EDC will be affected by the APIs when CSL_EDC_L1P is used
CSL_EDC_L2	L2's EDC will be affected by the APIs when CSL_EDC_L2 is used

6.4.2 CSL_EdcClrAccessType

enum CSL_EdcClrAccessType
Specifies the Access Type for which the parity error is to be cleared. Used to indicate which access parity error bit to be cleared.

Enumeration values:

CSL_EDC_DCLR	Data fetch parity error bit to be cleared
CSL_EDC_PCLR	Program fetch parity error bit to be cleared
CSL_EDC_DMACLR	DMA read parity error bit to be cleared
CSL_EDC_CECNTCLR	Correctable parity error count value to be cleared
CSL_EDC_NCECNTCLR	Non-correctable parity error count value to be cleared

6.4.3 CSL_EdcHwStatusQuery

enum CSL_EdcHwStatusQuery
EDC Hardware Status Query Type. Used to indicate what HW status to query.

Enumeration values:

CSL_EDC_QUERY_ENABLESTAT	Query enabled/disabled status. **Parameters:** *(CSL_EdcEnableStatus)*
CSL_EDC_QUERY_ERRORSTAT	Query error status. **Parameters:** *(CSL_EdcErrorStatus)*
CSL_EDC_QUERY_NERRSTAT	Query number of bit error status (L2 only). **Parameters:** *(CSL_EdcNumErrors)*
CSL_EDC_QUERY_BITPOS	Query bit position of error (L2 only). **Parameters:** *(Uint32 *)*
CSL_EDC_QUERY_ALLSTAT	Query all status (returns all bit fields EDSTAT

	register). **Parameters:** *CSL_EdcStatusInfo**)
CSL_EDC_QUERY_PAGE0	Query page 0 enables (L2 only). **Parameters:** *(Uint32 **)
CSL_EDC_QUERY_PAGE1	Query page 1 enables (L2 only). **Parameters:** *(Uint32 **)
CSL_EDC_QUERY_CE_CNT	Query correctable error count (L2 only). **Parameters:** *(Uint32 **)
CSL_EDC_QUERY_NCE_CNT	Query non-correctable error count (L2 only). **Parameters:** *(Uint32 **)

6.4.4 CSL_EdcEnableStatus

enum CSL_EdcEnableStatus
EDC Enable/Disable Status. Used to indicate whether EDC is enabled, disabled, or suspended.

Enumeration values:

CSL_EDC_ENABLED	EDC enabled
CSL_EDC_DISABLED	EDC disabled
CSL_EDC_SUSPENDED	EDC suspended

6.4.5 CSL_EdcErrorStatus

enum CSL_EdcErrorStatus
EDC error status. Used to indicate EDC access error type.

Enumeration values:

CSL_EDC_DERR	EDC error status - data fetch parity error
CSL_EDC_IERR	EDC error status - program fetch parity error
CSL_EDC_DMAERR	EDC error status - DMA read parity error

6.4.6 CSL_EdcNumErrors

enum CSL_EdcNumErrors
Indicates the number of EDC bit errors. Used to indicate number of EDC bit errors or if bit error is in parity value.

Enumeration values:

CSL_EDC_1BIT	EDC number of bit errors - single bit error
CSL_EDC_2BIT	EDC number of bit errors - double bit error
CSL_EDC_PERROR	EDC number of bit errors - error in parity value

6.4.7 CSL_EdcUmap

enum CSL_EdcUmap
UMAP Specifier for EDC. Used to indicate which of the UMAPs the page enables are to be applied to.

Enumeration values:

CSL_EDC_UMAP0	EDC apply page enables to UMAP0 only
CSL_EDC_UMAP1	EDC apply page enables to UMAP1 only
CSL_EDC_UMAPBOTH	EDC apply page enables to both UMAP0 and UMAP1

6.4.8 CSL_EdcAddrL2way

enum CSL_EdcAddrL2way
L2 way specifier for EDC error. Provides the L2 way for the Address of the detected error.

Enumeration values:

CSL_EDC_L2WAY_0	L2 way 0
CSL_EDC_L2WAY_1	L2 way 1
CSL_EDC_L2WAY_2	L2 way 2
CSL_EDC_L2WAY_3	L2 way 3

6.4.9 CSL_EdcAddrSram

enum CSL_EdcAddrSram
EDC error in SRAM or Cache. Specifies whether the EDC error was located in the SRAM or Cache.

Enumeration values:

CSL_EDC_CACHE	EDC error is in cache
CSL_EDC_SRAM	EDC error is in SRAM

Chapter 7
EDMA Module

Topics

7.1 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 8 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.

7.2 Functions

This section lists the functions available in the EDMA module.

7.2.1 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) {
    exit;
}
```

7.2.2 CSL_edma3Open

CSL_Edma3Handle CSL_edma3Open (CSL_Edma3Obj * *edmaObj*,
CSL_InstNum *edmaNum*,
CSL_Edma3ModuleAttr * *attr*,
CSL_Status * *status*
)

Description
This function Opens the EDMA 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

```
edmaObj         Pointer to EDMA Module Object

edmaNum         Instance of EDMA to be opened

attr            EDMA Attribute pointer

status          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 status variable. If status returned is

- `CSL_SOK` Valid EDMA handle is returned
- `CSL_ESYS_FAIL` The EDMA instance is invalid
- `CSL_ESYS_INVPARAMS` The Parameter 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_Edma3Context    context;
CSL_Status          status;
// Module Initialization
CSL_edma3Init(&context);
// Module Level Open
hModule = CSL_edma3Open(&edmaObj,CSL_EDMA3,NULL,&status);
```

7.2.3 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 that order successfully 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,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3Context            context;
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(&context);
// 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
CSL_edma3Close(hModule);
```

7.2.4 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 this API can be invoked.

Post Condition

EDMA registers are configured according to the hardware setup parameters.

Modifies

EDMA registers

Example

```
    CSL_Edma3Handle             hModule;
    CSL_Edma3HwSetup            hwSetup,gethwSetup;
    CSL_Edma3Obj                edmaObj;
    CSL_Edma3QueryInfo          info;
    CSL_Edma3CmdIntr            regionIntr;
    CSL_Edma3CmdDrae            regionAccess;
    CSL_Edma3Context            context;
    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;
    Uint32                      i, passStatus = 1;


    // Module Initialization
    CSL_edma3Init(&context);

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

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

7.2.5 CSL_edma3GetHwSetup

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

Description
It gets the hwparameterssetup of the all edma/qdma channels.

Arguments

hMod	Edma Handle
setup	Pointer to the setup structure

Return Value
CSL_Status

- CSL_SOK - Getting the msparameters 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 that order before CSL_edma3GetHwSetup() can be called.

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_Edma3ChannelObj         ChObj;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Status                  status;
Uint32                      i, passStatus = 1;


// Module Initialization
CSL_edma3Init(&context);

// 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;
CSL_edma3GetHwSetup(hModule,&gethwSetup);
```

7.2.6 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 that order before this API can be invoked.

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,gethwSetup;
        CSL_Edma3Obj                edmaObj;
        CSL_Edma3QueryInfo          info;
        CSL_Edma3CmdIntr            regionIntr;
        CSL_Edma3CmdDrae            regionAccess;
        CSL_Edma3Context            context;
        CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
\                                   CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
        CSL_Status                  status;
        Uint32                      i,passStatus = 1;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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);
```

7.2.7 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 – Invalid parameters

Pre Condition
The functions CSL_edma3Init(), CSL_edma3Open() must be called successfully in that order before this API can be invoked. 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,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3QueryInfo          info;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                              CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
// Module Initialization
CSL_edma3Init(&context);
// 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);
```

7.2.8 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-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

```
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 Parameter

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);
```

7.2.9 CSL_edma3ChannelOpen

```
CSL_Edma3ChannelHandle CSL_edma3ChannelOpen( CSL_Edma3ChannelObj * edmaObj,
                                             CSL_InstNum          edmaNum,
                                             CSL_Edma3ChannelAttr * chAttr,
                                             CSL_Status *         status
                                           )
```

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

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

edmaNum         Instance of EDMA whose channel is requested

chAttr          Instance of Channel requested

status          Status of the function call
```

Return Value

CSL_Edma3ChannelHandle

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

Pre Condition

Functions CSL_edma3Init(), CSL_edma3Open() must be invoked successfully in that order before this API can be invoked.

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 or channel is invalid
- CSL_ESYS_INVPARAMS The Parameter passed is 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,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3QueryInfo          info;
CSL_Edma3CmdIntr            regionIntr;
CSL_Edma3CmdDrae            regionAccess;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ParamSetup         myParamSetup;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                               CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Status                  status;
Uint32                      i, passStatus = 1;

// Module Initialization
```

```
    CSL_edma3Init(&context);

    // 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_TEVT6L;
    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);
   regionIntr.region = CSL_EDMA3_REGION_0;
   regionIntr.intr = 0;
   regionIntr.intrh = 0;

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

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

7.2.10 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 that order before this API can be invoked.

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,gethwSetup;
        CSL_Edma3Obj            edmaObj;
        CSL_Edma3ParamHandle    hParamPing,hParamPong,hParamBasic;
        CSL_Edma3ChannelObj     chObj;
        CSL_Edma3QueryInfo      info;
        CSL_Edma3CmdIntr        regionIntr;
        CSL_Edma3CmdDrae        regionAccess;
        CSL_Edma3ChannelHandle  hChannel;
        CSL_Edma3ParamSetup     myParamSetup;
        CSL_Edma3Context        context;
        CSL_Edma3ChannelAttr    chAttr;
        CSL_Edma3HwDmaChannelSetup dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                                    CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup getdmahwSetup[CSL_EDMA3_NUM_DMACH];

        CSL_Status              status;
        Uint32                  i, passStatus = 1;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
        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);
        regionIntr.region = CSL_EDMA3_REGION_0;
        regionIntr.intr = 0;
        regionIntr.intrh = 0;

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

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

7.2.11 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 that order before this API can be invoked.

Post Condition
Channel to parameter entry is configured.

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
Uint16                      paramNum;
CSL_Status                  status;
Uint32                      i,passStatus = 1;


// Module Initialization
CSL_edma3Init(&context);

// 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_TEVT6L;
hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

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

7.2.12 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 that order before CSL_edma3HwChannelSetupTriggerWord() can be called.

Post Condition
None

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Status                  status;
Uint32                      i, passStatus = 1;


// Module Initialization
CSL_edma3Init(&context);

// 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 the 3rd trigger
                                                        word
CSL_edma3HwChannelSetupTriggerWord(hChannel,3);
```

7.2.13 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 that order before this API can be invoked.
Post Condition
Sets up the channel to Queue mapping

Modifies
EDMA registers

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                            CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Edma3Que                evtQue;
CSL_Status                  status;
```

```
        Uint32                      i, passStatus = 1;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
        hChannel = CSL_edma3ChannelOpen(&chObj,
                                CSL_EDMA3,
                                &chAttr,
                                &status);

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

7.2.14 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 that order before CSL_edma3GetHwChannelSetupParam() can be invoked.

Post Condition
None

Modifies
None

Example

```
        CSL_Edma3Handle             hModule;
        CSL_Edma3HwSetup            hwSetup,gethwSetup;
        CSL_Edma3Obj                edmaObj;
        CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
        CSL_Edma3ChannelObj         chObj;
        CSL_Edma3ChannelHandle      hChannel;
        CSL_Edma3Context            context;
        CSL_Edma3ChannelAttr        chAttr;
        CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                                     CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
        Uint16                      paramNum;
        CSL_Status                  status;
        Uint32                      i, passStatus = 1;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
        hChannel = CSL_edma3ChannelOpen(&chObj,
                                        CSL_EDMA3,
                                        &chAttr,
                                        &status);

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

7.2.15 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 that order before CSL_edma3GetHwChannelSetupTriggerWord() can be called.

Post Condition
None

Modifies
None

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                            CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
Uint8                       triggerWord;
CSL_Status                  status;
Uint32                      i, passStatus = 1;


// Module Initialization
CSL_edma3Init(&context);

// 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);
```

7.2.16 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 that order before CSL_edma3GetHwChannelSetupQue() can be called.

Post Condition
None

Modifies
None

Example

```
      CSL_Edma3Handle            hModule;
      CSL_Edma3HwSetup           hwSetup,gethwSetup;
      CSL_Edma3Obj               edmaObj;
      CSL_Edma3ParamHandle       hParamPing, hParamPong, hParamBasic;
```

```
CSL_Edma3ChannelObj         chObj;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                             CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
CSL_Edma3Que                evtQue;
CSL_Status                  status;
Uint32                      i, passStatus = 1;
// Module Initialization
CSL_edma3Init(&context);

// 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_TEVT6L;
  hChannel = CSL_edma3ChannelOpen(&ChObj,
                          CSL_EDMA3,
                          &chAttr,
                          &status);

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

7.2.17 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 that order before this API can be invoked. If a Shadow region is used, then care of the DRAE settings must be taken.

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,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3CmdIntr            regionIntr;
CSL_Edma3CmdDrae            regionAccess;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
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(&context);

// 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_TEVT6L;
       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);
```

7.2.18 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

Pre Condition
The functions CSL_edma3Init(), CSL_edma3Open() and CSL_edma3ChannelOpen() must be called successfully in that order before this API can be invoked. If a Shadow region is used, then care of the DRAE settings must be taken.

Post Condition
None

Modifies
The input argument "response" is modified.

Example

```
        CSL_Edma3Handle             hModule;
        CSL_Edma3HwSetup            hwSetup,gethwSetup;
        CSL_Edma3Obj                edmaObj;
        CSL_Edma3ParamHandle        hParamPing,hParamPong,hParamBasic;
        CSL_Edma3ChannelObj         chObj;
        CSL_Edma3QueryInfo          info;
        CSL_Edma3CmdIntr            regionIntr;
        CSL_Edma3CmdDrae            regionAccess;
        CSL_Edma3ChannelHandle      hChannel;
        CSL_Edma3ParamSetup         myParamSetup;
        CSL_Edma3Context            context;
        CSL_Edma3ChannelAttr        chAttr;
        CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] = \
                                     CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
        CSL_Status                  status;
        Bool                        errStat;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
        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);
```

7.2.19 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 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.
- Returns NULL if the channel handle is invalid and sets the status to CSL_ESYS_BADHANDLE.
- Returns NULL if the param number is invalid and sets the status to CSL_ESYS_INVPARAMS.

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

Post Condition
None

Modifies
None

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3QueryInfo          info;
CSL_Edma3CmdIntr            regionIntr;
CSL_Edma3CmdDrae            regionAccess;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ParamSetup         myParamSetup;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
```

```
                                CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
        CSL_Status                  status;
        Uint32                      i,passStatus = 1;
        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
         hChannel = CSL_edma3ChannelOpen(&chObj,
                                 CSL_EDMA3,
                                 &chAttr,
                                 &status);

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

7.2.20 CSL_edma3ParamSetup

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

Description
This function configures the EDMA parameter 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 that order before this API can be invoked.

Post Condition
Configures the EDMA parameter entry

Modifies
Parameter entry

Example

```
        CSL_Edma3Handle             hModule;
        CSL_Edma3HwSetup            hwSetup,gethwSetup;
        CSL_Edma3Obj                edmaObj;
        CSL_Edma3ParamHandle        hParamPing;
        CSL_Edma3ChannelObj         chObj;
        CSL_Edma3QueryInfo          info;
        CSL_Edma3CmdIntr            regionIntr;
        CSL_Edma3CmdDrae            regionAccess;
        CSL_Edma3ChannelHandle      hChannel;
        CSL_Edma3ParamSetup         myParamSetup;
        CSL_Edma3Context            context;
        CSL_Edma3ChannelAttr        chAttr;
        CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                                     CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
        CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];
        CSL_Status                  status;
        Uint32                      i, passStatus = 1;


        // Module Initialization
        CSL_edma3Init(&context);

        // 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_TEVT6L;
        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);
```

7.2.21 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 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 - Invlaid handle
- CSL_ESYS_INVPARAMS The Parameter passed is invalid

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

Post Condition
Configure trigger word

Modifies
None

Example

```
CSL_Edma3Handle             hModule;
CSL_Edma3HwSetup            hwSetup,gethwSetup;
CSL_Edma3Obj                edmaObj;
CSL_Edma3ParamHandle        hParamPing;
CSL_Edma3ChannelObj         chObj;
CSL_Edma3QueryInfo          info;
CSL_Edma3CmdIntr            regionIntr;
CSL_Edma3CmdQrae            regionAccess;
CSL_Edma3ChannelHandle      hChannel;
CSL_Edma3ParamSetup         myParamSetup;
CSL_Edma3Context            context;
CSL_Edma3ChannelAttr        chAttr;
CSL_Edma3HwDmaChannelSetup  dmahwSetup[CSL_EDMA3_NUM_DMACH] =
                            CSL_EDMA3_DMACHANNELSETUP_DEFAULT;
CSL_Edma3HwQdmaChannelSetup qdmahwSetup[CSL_EDMA3_NUM_QDMACH] =
                            CSL_EDMA3_QDMACHANNELSETUP_DEFAULT;
CSL_Edma3HwDmaChannelSetup  getdmahwSetup[CSL_EDMA3_NUM_DMACH];

// Module Initialization
CSL_edma3Init(&context);

// 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_CHA_TEVT6L;
   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);
```

7.3 Data Structures

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

7.3.1 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.

7.3.2 CSL_Edma3ParamSetup

Detailed Description
Edma ParamSetup 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 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 Upper 16 bits are b count reload

Uint32 CSL_Edma3ParamSetup::option
Options

Uint32 CSL_Edma3ParamSetup::srcAddr
Specifies the source address

Uint32 CSL_Edma3ParamSetup::srcDstBidx
Lower 16 bits are source b index Upper 16 bits are destination b index

Uint32 CSL_Edma3ParamSetup::srcDstCidx
Lower 16 bits are source c index Upper 16 bits are destination c index

7.3.3 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

7.3.4 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
Whether number of permissible outstanding TCCs is exceeded

7.3.5 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

7.3.6 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

7.3.7 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

7.3.8 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

7.3.9 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

7.3.10 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

7.3.11 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 when

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

7.3.12 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

7.3.13 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

7.3.14 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

7.3.15 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

7.3.16 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.

7.3.17 CSL_Edma3HwQdmaChannelSetup

Detailed Description
QDMA 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

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

CSL_Edma3Que CSL_Edma3HwQdmaChannelSetup::que
Queue number for the channel

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

7.3.18 CSL_Edma3HwDmaChannelSetup

Detailed Description
QDMA 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
Parameter set mapping for the channel

7.3.19 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

7.3.20 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. This is relevant only is MPEXIST is present for a given device.

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

7.4 Enumerations

7.4.1 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

7.4.2 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

7.4.3 CSL_Edma3HwControlCmd

enum CSL_Edma3HwControlCmd
MODULE Level Commands

Enumeration values:

CSL_EDMA3_CMD_MEMPROTECT_SET	(Arg: *CSL_Edma3CmdRegion**) Programmation of MPPAG,MPPA[0-7] attributes
CSL_EDMA3_CMD_MEMFAULT_CLEAR	(Arg: None)Clear Memory Fault
CSL_EDMA3_CMD_DMAREGION_ENABLE	(Arg: *CSL_Edma3CmdDrae**) 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
CSL_EDMA3_CMD_DMAREGION_DISABLE	(Arg: *CSL_Edma3CmdDrae**) Disables bits as specified in the argument passed in DRAE/DRAEH
CSL_EDMA3_CMD_QDMAREGION_ENABLE	(Arg: *CSL_Edma3CmdQrae**) 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.
CSL_EDMA3_CMD_QDMAREGION_DISABLE	(Arg: *CSL_Edma3CmdQrae**)Disables bits as specified in the argument passed in QRAE
CSL_EDMA3_CMD_QUEPRIORITY_SET	(Arg: *CSL_Edma3CmdQuePri**) Programming QUEPRI register with the specified priority
CSL_EDMA3_CMD_QUETHRESHOLD_SET	(Arg: *CSL_Edma3CmdQueThr**) Programming QUEUE Threshold levels
CSL_EDMA3_CMD_ERROR_EVAL	(Arg: #None)Sets the EVAL bit in the EEVAL register
CSL_EDMA3_CMD_INTRPEND_CLEAR	(Arg: *CSL_Edma3CmdIntr**) Clears specified (Bitmask) pending interrupt at Module/Region Level
CSL_EDMA3_CMD_INTR_ENABLE	(Arg: *CSL_Edma3CmdIntr**)Enables specified interrupts (BitMask) at Module/Region Level
CSL_EDMA3_CMD_INTR_DISABLE	(Arg: *CSL_Edma3CmdIntr**)Disables specified interrupts (BitMask) at Module/Region Level
CSL_EDMA3_CMD_INTR_EVAL	(Arg: #*Int**) Interrupt Evaluation asserted for the Module/Region
CSL_EDMA3_CMD_CTRLERROR_CLEAR	(Arg: *CSL_Edma3CtrlErrStat**)
CSL_EDMA3_CMD_EVENTMISSED_CLEAR	(Arg: #*CSL_BitMask32**) 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.

7.4.4 CSL_Edma3HwStatusQuery

enum CSL_Edma3HwStatusQuery
MODULE Level Queries.

Enumeration values:

CSL_EDMA3_QUERY_MEMFAULT	(Arg: *CSL_Edma3MemFaultStat**)Return the Memory fault details
CSL_EDMA3_QUERY_MEMPROTECT	Arg: *CSL_Edma3CmdRegion**)Return memory attribute of the specified region
CSL_EDMA3_QUERY_CTRLERROR	(Arg: *CSL_Edma3CtrlErrStat**)Return Controller Error
CSL_EDMA3_QUERY_INTRPEND	(Arg: *CSL_Edma3CmdIntr**)Return pend status of specified interrupt
CSL_EDMA3_QUERY_EVENTMISSED	(Arg: *#CSL_BitMask32**)Returns Miss 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
CSL_EDMA3_QUERY_QUESTATUS	(Arg: *CSL_Edma3QueStat**)Returns the Que status
CSL_EDMA3_QUERY_ACTIVITY	(Arg: *CSL_Edma3ActivityStat**)Returns the Channel Controller Active Status
CSL_EDMA3_QUERY_INFO	(Arg: *CSL_Edma3QueryInfo**)Returns the Channel Controller Information viz. Configuration, Revision Id

7.4.5 CSL_Edma3HwChannelControlCmd

enum CSL_Edma3HwChannelControlCmd
CHANNEL Commands.

Enumeration values:

CSL_EDMA3_CMD_CHANNEL_ENABLE	(Arg: #None)Enables specified Channel
CSL_EDMA3_CMD_CHANNEL_DISABLE	(Arg: #None)Disables specified Channel
CSL_EDMA3_CMD_CHANNEL_SET	(Arg: #None)Manually sets the Channel Event,writes into ESR/ESRH and not ER.NA for QDMA.
CSL_EDMA3_CMD_CHANNEL_CLEAR	(Arg: #None)Manually clears the Channel Event, does not write into ESR/ESRH or ER/ERH but the ECR/ECRH. NA for QDMA.
CSL_EDMA3_CMD_CHANNEL_CLEARERR	(Arg: CSL_Edma3ChannelErr*)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)

7.4.6 CSL_Edma3HwChannelStatusQuery

enum CSL_Edma3HwChannelStatusQuery
CHANNEL Queries.

Enumeration values:

CSL_EDMA3_QUERY_CHANNEL_STATUS	(Arg: *#Bool**)In case of DMA channels returns TRUE if ER/ERH is set, In case of QDMA channels returns TRUE if QER is set
CSL_EDMA3_QUERY_CHANNEL_ERR	(Arg: *CSL_Edma3ChannelErr**)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

7.5 Macros

#define CSL_EDMA3_ADDRMODE_CONST 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)

Value:
```
(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

#define CSL_EDMA3_CIDX_MAKE (src, dst)

Value:
```
(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

#define CSL_EDMA3_CNT_MAKE (aCnt, bCnt)

Value:
```
(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

#define CSL_EDMA3_DMACHANNELSETUP_DEFAULT
Value:
```
{       \
   {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_128BIT 4

128 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_16BIT 1
16 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_256BIT 5
256 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_8BIT 0
8 bit FIFO Width

#define CSL_EDMA3_FIFOWIDTH_NONE 0
Only for ease

#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)

Value:

```
(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

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

Value:

```
(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
Value:

```
{       \
   {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 set

#define CSL_EDMA3_TRIGWORD_OPT 0
Channel Options.

#define CSL_EDMA3_TRIGWORD_SRC 1
Channel Source address

#define CSL_EDMA3_TRIGWORD_A_B_CNT 2
Count for 1st Dimension and 2nd Dimension.

#define CSL_EDMA3_TRIGWORD_DST 3
Channel Destination address.

#define CSL_EDMA3_TRIGWORD_SRC_DST_BIDX 4
Source BCNT & Destination BCNT index.

#define CSL_EDMA3_TRIGWORD_LINK_BCNTRLD 5
Link address and BCNT reload.

#define CSL_EDMA3_TRIGWORD_SRC_DST_CIDX 6
Source CCNT index and Destination CCNT index.

#define CSL_EDMA3_TRIGWORD_CCNT 7
Count for 3rd Dimension.

#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’

7.6 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 struct CSL_Edma3Obj * CSL_Edma3Handle
EDMA handle.

Typedef volatile CSL_Edma3ccParamsetRegs * CSL_Edma3ParamHandle
CSL Parameter Set Handle.

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

Chapter 8
EMAC Module

Topics

8.1 Overview

The Ethernet Media Access Controller (EMAC) module provides an interface between the C6472 DSP core processor and the networked community. The EMAC supports 10Base-T (10 Mbits/second [Mbps]), and 100BaseTX (100 Mbps), in either half- or full-duplex mode, and 1000BaseT (1000 Mbps) in full-duplex mode, with hardware flow control and quality-of-service (QOS) support. There are two ports of EMAC on C6472 DSP.

The EMAC module conforms to the IEEE 802.3-2002 standard, describing the “Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer” specifications. The IEEE 802.3 standard has also been adopted by ISO/IEC and re-designated as ISO/IEC 8802-3:2000(E). Deviation from this standard, the EMAC module does not use the Transmit Coding Error signal MTXER. Instead of driving the error pin when an underflow condition occurs on a transmitted frame, the EMAC will intentionally generate an incorrect checksum by inverting the frame CRC, so that the transmitted frame will be detected as an error by the network.

The EMAC control module is the main interface between the device core processor, the MDIO module, and the EMAC module. The EMAC control module contains the necessary components to allow the EMAC to make efficient use of device memory, plus it controls device interrupts. The EMAC control module incorporates 8K-bytes of internal RAM to hold EMAC buffer descriptors.

8.2 Functions

This section lists Functions available in the EMAC module.

8.2.1 EMAC_open

Uint32 EMAC_open	**(**	**int**	**instNum,**
		Handle	**hApplication,**
		EMAC_Config *	**pEMACConfig,**
		Handle	**hEMAC**
		Handle	**hCore**
	)		

Description

Opens the EMAC peripheral at the given physical index and initializes it to an embryonic state.

The calling application must supply operating configurations that includes the ones common to all the cores if on the master core and also those specific to individual cores. The EMAC device must be closed and then re-opened when new configuration is required.

The application layer may pass in an hApplication callback handle, that will be supplied by the EMAC device when making calls to the application callback functions.

A valid EMAC device handle should be passed to this API to which the configuration and operating state of the EMAC device will be written.

A valid EMAC core instance handle should be passed to this API to which the operating state of the EMAC device specific to individual cores will be written.

The default receive filter prevents normal packets from being received until the receive filter is specified by calling **EMAC_receiveFilter()**.

By calling **EMAC_close()** followed by **EMAC_open()** on the master core, EMAC device reset is achieved. When the same sequence is called on a non-master core, a core reset for EMAC operation is achieved

The function returns zero on success, or an error code on failure.

Possible error codes include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

instNum	EMAC Peripheral ID to identify the EMAC controller to be initialized.
hApplication	Application handle
pEMACConfig	EMAC's configuration structure
hEMAC	Handle to the EMAC device which needs to be initialized.
hCore	Handle to the EMAC core instance which needs to be initialized.

Return Value
Uint32

- 0 – Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
None

Post Condition
Opens the EMAC peripheral at the given physical index and initializes it.

Modifies
EMAC configuration registers

Example

```
#define MASTER_CORE 0          // Which core is responsible for
                                  EMAC common initialization

EMAC_Device            EMACObj;
// In multi-core scenario, this must be placed in shared
   memory, and initialized by application.
EMAC_Core              EMACCore;
// In multi-core scenario, it is desirable to place EMACCore in
   local memory so that run-time cache operation on EMACCore
   can be avoided.

Uint32                 coreNum;
Uint32                 i = 0, j = 0;
EMAC_Config            ecfg;
EMAC_AddrConfig*       addrCfg;

// Initialize our EMAC Dev structure.
if(coreNum == MASTER_CORE)
{
    memset(&EMACObj, 0, sizeof(EMAC_Device));

    // Not using the MDIO configuration
    ecfg.EMACCommonConfig.UseMdio = 0;

    // core 0 is master core performing common initialization
       of the EMAC
    ecfg.EMACCommonConfig.CoreNum = 0;

    //packet size
    ecfg.EMACCommonConfig.PktMTU = 1600;

    // Setup the EMAC local loopback
    ecfg.EMACCommonConfig.ModeFlags      =
    EMAC_CONFIG_MODEFLG_MACLOOPBACK |
    EMAC_CONFIG_MODEFLG_GMIIEN |
    EMAC_CONFIG_MODEFLG_FULLDUPLEX;

    ecfg.EMACCommonConfig.MdioModeFlags  = MDIO_MODEFLG_FD1000;
```

```
}

// Initialize EMAC core instance structure.
memset(&EMACCore, 0, sizeof(EMAC_Core));

//Total 3 MAC addresses allocated for the receive channel
ecfg.EMACCoreConfig.NumOfMacAddrs = 3;
// selects CPPI RAM for Descriptor memory
ecfg.EMACCoreConfig.DescBase = EMAC_DESC_BASE_CPPI;

ecfg.EMACCoreConfig.RxMaxPktPool          = 8;
ecfg.EMACCoreConfig.pfcbGetPacket         = &GetPacket;
ecfg.EMACCoreConfig.pfcbFreePacket        = &FreePacket;
ecfg.EMACCoreConfig.pfcbRxPacket          = &RxPacket;
ecfg.EMACCoreConfig.pfcbStatus            = &StatusUpdate;
ecfg.EMACCoreConfig.pfcbStatistics        = &StatisticsUpdate;

switch(coreNum) {
    default:
    case 0:                   // core 0 use channel 0
        ecfg.EMACCoreConfig.ChannelInfo.TxChanEnable = 1;
        ecfg.EMACCoreConfig.ChannelInfo.RxChanEnable = 1;
        break;
    case 1:                   // core 1 use channel 1
        ecfg.EMACCoreConfig.ChannelInfo.TxChanEnable = 2;
        ecfg.EMACCoreConfig.ChannelInfo.RxChanEnable = 2;
        break;
    case 2:                   // core 2 use channel 2
        ecfg.EMACCoreConfig.ChannelInfo.TxChanEnable = 4;
        ecfg.EMACCoreConfig.ChannelInfo.RxChanEnable = 4;
        break;
}
//Configure the number of MAC addresses per channel
//Hardware gives support for 32 MAC addresses for 8 receive
//channels
//Here total 9 MAC addresses are assigned to 3 receive channels
//3 MAC addresses per channel
//MAC addresses and channels allocated are like mentioned below
//  core no        channel assigned       MAC address
//   core0         channel 0           00.01.02.03.04.05
//                                     10.11.12.13.14.15
//                                     20.21.22.23.24.25
//                         (address used for loopback test)
//   core1         channel 1           30.31.32.33.34.35
//                                     40.41.42.43.44.45
//                                     50.51.52.53.54.55
//                         (address used for loopback test)
//   core2         channel 2           60.61.62.63.64.65
//                                     70.71.72.73.74.75
//                                     80.81.82.83.84.85
//                         (address used for loopback test)

ecfg.EMACCoreConfig.MacAddr = (EMAC_AddrConfig **) malloc
(ecfg.EMACCoreConfig.NumOfMacAddrs * sizeof(EMAC_AddrConfig *));

for (j=0; j<ecfg.EMACCoreConfig.NumOfMacAddrs; j++){
```

```
        ecfg.EMACCoreConfig.MacAddr[j] =
        (EMAC_AddrConfig*)malloc(sizeof(EMAC_AddrConfig));
    }

    for(j=0; (Uint8)j<(ecfg.EMACCoreConfig.NumOfMacAddrs); j++){
        addrCfg = ecfg.EMACCoreConfig.MacAddr[j];
        addrCfg->ChannelNum = coreNum;
        for (i=0; i<6; i++)
        {
            addrCfg->Addr[i] = j * 0x10 + i + coreNum * 0x30;
        }
    }

    EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
```

8.2.2 EMAC_commonInit()

Uint32 EMAC_commonInit() (int instNum, EMAC_Common_Config * pEMACCommonConfig, Handle hEMAC)

Description

Open the EMAC peripheral at the given physical index and perform initialization common to all the cores, including resetting the EMAC control module and stats, initializing DMA, configuring common registers such as the MACCONTROL register, initializing MDIO, and etc. This function has to be called only once by the master core.

The calling application must supply an operating configuration. Data from this config structure is copied into the device's internal instance structure so the structure may be discarded after **EMAC_commonInit()** returns. In order to change an item in the configuration, the EMAC device must be de-initialized by **EMAC_commonDeInit()** and then re-initialized by calling EMAC_commonInit() with the new configuration.

A valid EMAC device handle should be passed to this API to which the configuration and operating state of the EMAC device common to all the cores will be written. The EMAC device structure for the handle needs to be initialized before calling **EMAC_commonInit()**.

The default receive filter prevents normal packets from being received until the receive filter is specified by calling **EMAC_receiveFilter()**.

A device reset is achieved by calling **EMAC_commonDeInit()** followed by **EMAC_commonInit()**.

The function returns zero on success, or an error code on failure.

Possible error code include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

```
instNum           EMAC Peripheral ID to identify the EMAC
                  controller to be initialized.
pEMACCommonConfig EMAC's configuration common to all cores.
hEMAC             Handle to the EMAC device which needs to be
                  initialized.
```

Return Value
`Uint32`

- `0` – Success
- `EMAC_ERROR_INVALID` - A calling parameter is invalid

Pre Condition
None

Post Condition
Open the EMAC peripheral at the given physical index and perform initialization common to all the cores.

Example

```
EMAC_Device            EMACObj;
// In multi-core scenario, this must be placed in shared
   memory, and initialized by application. Refer to
   "emac_core_restart" example project for more details.

EMAC_Common_Config     commonCfg;

// Initialize our EMAC Dev structure.
memset(&EMACObj, 0, sizeof(EMAC_Device));

// Not using the MDIO configuration
commonCfg.UseMdio = 0;

// core 0 is master core performing common initialization of
   the EMAC
commonCfg.CoreNum = 0;

//packet size
commonCfg.PktMTU = 1600;

// Setup the EMAC local loopback
commonCfg.ModeFlags      = EMAC_CONFIG_MODEFLG_MACLOOPBACK|
                           EMAC_CONFIG_MODEFLG_GMIIEN|
                           EMAC_CONFIG_MODEFLG_FULLDUPLEX;

commonCfg.MdioModeFlags  = MDIO_MODEFLG_FD1000;

//Common initialization of the EMAC peripheral
EMAC_commonInit(0, &commonCfg, &EMACObj);
```

8.2.3 EMAC_coreInit()

Uint32 EMAC_coreInit()	**(Handle**	**hEMAC,**
	Handle	**hApplication,**
	EMAC_Core_Config *	**pEMACCoreConfig,**
	Handle	**hCore**

)

Description
Core specific initialization to use the EMAC peripheral at the given physical index when every core starts/restarts. The per core configuration includes setting up Tx/Rx channels, allocating MAC addresses, enabling interrupts in the EMAC control module for the channels that this core uses, and etc.

The calling application must supply an operating configuration. Data from this config structure is copied into the device's internal instance structure so the structure may be discarded after **EMAC_coreInit()** returns. In order to change an item in the configuration, the core must be de-initialized from the EMAC device by calling **EMAC_coreDeInit()** and then re-initialized by calling EMAC_coreInit() with the new configuration.

The application layer may pass in an hApplication callback handle, that will be supplied by the EMAC device for the core when making calls to the core's application callback functions.

A valid EMAC device handle should be passed to this API to which the per core configuration and operating state of the EMAC device will be written.

A valid EMAC core instance handle should be passed to this API to which the operating state of the EMAC device specific to individual cores will be written. The EMAC core instance structure for the handle needs to be initialized before calling EMAC_coreInit().

A core reset for EMAC operation is achieved by calling **EMAC_coreDeInit()** followed by **EMAC_coreInit()**.

The function returns zero on success, or an error code on failure.

Possible error code include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

```
hEMAC      Handle to the EMAC device which needs to be
           initialized.
hApplication    Application handle
pEMACCoreConfig EMAC's configuration specific to the core
hCore           Handle to the EMAC core instance which needs to
                be initialized.
```

Return Value
`Uint32`

- `0` – Success
- `EMAC_ERROR_INVALID` - A calling parameter is invalid

Pre Condition
Before calling this API, EMAC_commonInit function must be called on the master core and then EMAC_coreDeInit must be called for this particular core.

Post Condition
Core specific initialization to use the EMAC peripheral at the given physical index.

Example

```
#define MASTER_CORE 0           //Which core is responsible for
                                  EMAC common initialization

Uint32                coreNum;
Uint32                i = 0, j = 0;
EMAC_Common_Config    commonCfg;
EMAC_Core_Config      coreCfg;
EMAC_Device           EMACObj;
EMAC_Core             EMACCore;
EMAC_AddrConfig*      addrCfg;

...

if (coreNum == MASTER_CORE)
{
    // Initialize our EMAC Dev structure.
    memset(&EMACObj, 0, sizeof(EMAC_Device));

    // Common initialization of the EMAC peripheral
    EMAC_commonInit(0, &commonCfg, &EMACObj);
}

// Initialize EMAC core instance structure.
memset(&EMACCore, 0, sizeof(EMAC_Core));

//Total 3 MAC addresses allocated for the receive channel
coreCfg.NumOfMacAddrs = 3;
// selects CPPI RAM for Descriptor memory
coreCfg.DescBase = EMAC_DESC_BASE_CPPI;

coreCfg.RxMaxPktPool             = 8;
coreCfg.pfcbGetPacket            = &GetPacket;
coreCfg.pfcbFreePacket           = &FreePacket;
coreCfg.pfcbRxPacket             = &RxPacket;
coreCfg.pfcbStatus               = &StatusUpdate;
coreCfg.pfcbStatistics           = &StatisticsUpdate;

switch(coreNum) {
    default:
    case 0:                  // core 0 use channel 0
        coreCfg.ChannelInfo.TxChanEnable = 1;
        coreCfg.ChannelInfo.RxChanEnable = 1;
        break;
    case 1:                  // core 1 use channel 1
        coreCfg.ChannelInfo.TxChanEnable = 2;
        coreCfg.ChannelInfo.RxChanEnable = 2;
        break;
    case 2:                  // core 2 use channel 2
        coreCfg.ChannelInfo.TxChanEnable = 4;
        coreCfg.ChannelInfo.RxChanEnable = 4;
        break;
}

//Configure the number of MAC addresses per channel
//Hardware gives support for 32 MAC addresses for 8 receive
//channels
```

```
//Here total 9 MAC addresses are assigned to 3 receive channels
//3 MAC addresses per channel
//MAC addresses and channels allocated are like mentioned below
//  core no        channel assigned       MAC address
//   core0         channel 0           00.01.02.03.04.05
//                                     10.11.12.13.14.15
//                                     20.21.22.23.24.25
//                         (address used for loopback test)
//   core1         channel 1           30.31.32.33.34.35
//                                     40.41.42.43.44.45
//                                     50.51.52.53.54.55
//                         (address used for loopback test)
//   core2         channel 2           60.61.62.63.64.65
//                                     70.71.72.73.74.75
//                                     80.81.82.83.84.85
//                         (address used for loopback test)

coreCfg.MacAddr = (EMAC_AddrConfig **)
                  malloc(coreCfg.NumOfMacAddrs *
                  sizeof(EMAC_AddrConfig *));

for (j=0; j<coreCfg.NumOfMacAddrs; j++){
    coreCfg.MacAddr[j] = (EMAC_AddrConfig *)
                          malloc(sizeof(EMAC_AddrConfig));
}

for(j=0; (Uint8)j<(coreCfg.NumOfMacAddrs); j++){
    addrCfg = coreCfg.MacAddr[j];
    addrCfg->ChannelNum = coreNum;
    for (i=0; i<6; i++)
    {
        addrCfg->Addr[i] = j * 0x10 + i + coreNum * 0x30;
    }
}

EMAC_coreDeInit(&EMACObj);
EMAC_coreInit(&EMACObj, (Handle)0x12345678, &coreCfg,
              &EMACCore);
```

8.2.4 EMAC_close

Uint32 EMAC_close (Handle hEMAC)

Description

When called by a core, free all pending transmit and receive packets, and release Tx/Rx channels used and MAC addresses assigned via.calling **EMAC_coreDeInit()**. When called by the master core, furhter close the EMAC peripheral indicated by the supplied EMAC device handle, shutdown both send and receive operations, and bring down the MDIO link via calling **EMAC_commonDeInit()**. The function returns zero on success, or an error code on failure.

Possible error code include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

```
hEMAC       handle to opened the EMAC device
```

Return Value
Uint32

- 0 - Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC_open, or EMAC_commonInit/EMAC_coreInit function must be called before calling this API.

Post Condition
For any core, the EMAC device will free all pending transmit and receive packets, release Tx/Rx channels used and MAC addresses assigned. For master core, the EMAC device will further shutdown both send and receive operations and bring down the MDIO link.

Example

```
EMAC_Config  ecfg;
EMAC_Device  EMACObj;
EMAC_Core       EMACCore;
...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

//Close the EMAC peripheral
EMAC_close( &EMACObj );
```

8.2.5 EMAC_commonDeInit

Uint32 EMAC_commonDeInit (Handle hEMAC)

Description
Shut down EMAC peripheral indicated by the supplied EMAC device handle: tear down both send and receive operations, resets common configuration, and bring down the MDIO link. Called by the master core only. The function returns zero on success, or an error code on failure.

Possible error code include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

```
hEMAC       handle to opened the EMAC device
```

Return Value
Uint32

- 0 - Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC_open (on the master core) or EMAC_commonInit function must be called before calling this API.

Post Condition
The EMAC device will shutdown both send and receive operations, resets common configuration, and bring down the MDIO link.

Example

```
EMAC_Config     ecfg;
EMAC_Device     EMACObj;
EMAC_Core       EMACCore;
...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
...

//Shut down EMAC operation for a core
EMAC_coreDeInit( &EMACObj );

...

//Shut down the EMAC peripheral
EMAC_commonDeInit( &EMACObj );
```

8.2.6 EMAC_coreDeInit

Uint32 EMAC_coreDeInit (Handle hEMAC)

Description
Shut down EMAC operation for a core. When called, free all pending transmit and receive packets for the core, and release Tx/Rx channels used and MAC addresses assigned The function returns zero on success, or an error code on failure.

Possible error code include: EMAC_ERROR_INVALID - A calling parameter is invalid

Arguments

hEMAC	handle to opened the EMAC device

Return Value
Uint32

- 0 - Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC_open (on the master core) or EMAC_commonInit function must be called before calling this API.

Post Condition
The EMAC device will free all pending transmit and receive packets for the core, and also release Tx/Rx channels used and MAC addresses assigned.

Example

```
EMAC_Config     ecfg;
EMAC_Device     EMACObj;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

...

//Shut down EMAC operation for a core
EMAC_coreDeInit( &EMACObj );
```

8.2.7 EMAC_enumerate

Uint32 EMAC_enumerate (void)

Description
Enumerates the EMAC peripherals installed in the system and returns an integer count. The EMAC devices are enumerated in a consistent fashion so that each device can be later referenced by its physical index value ranging from "1" to "n" where "n" is the count returned by this function.

Return Value
Uint32

Pre Condition
None

Post Condition
None

Modifies
None

Example

```
EMAC_enumerate( );
```

8.2.8 EMAC_getReceiveFilter

Uint32 EMAC_getReceiveFilter (Handle hEMAC, Uint32 * pReceiveFilter)

Description
Called to get the current packet filter setting for received packets. The filter values are the same as those used in **EMAC_setReceiveFilter()**.The current filter value is written to the pointer supplied in pReceiveFilter.
Arguments

```
hEMAC             handle to the opened EMAC device

pReceiveFilter    Current receive packet filter
```

Return Value
`Uint32`

- `0` - Success
- `EMAC_ERROR_INVALID` - A calling parameter is invalid

Pre Condition
EMAC peripheral instance must be opened before calling this API and must be set the packet filter value.

Post Condition
The current filter value is written to the pointer supplied

Modifies
Output parameter *pReceiveFilter* being passed

Example

```
#define EMAC_RXFILTER_DIRECT       1
EMAC_Config     ecfg;
Uint32          pReceiveFilter;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
...

EMAC_setReceiveFilter(&EMACObj, EMAC_RXFILTER_DIRECT, 0);

EMAC_getReceiveFilter(&EMACObj, &pReceiveFilter );
```

8.2.9 EMAC_getStatistics

Uint32 EMAC_getStatistics (Handle hEMAC, EMAC_Statistics * pStatistics)

Description
Called to get the current device statistics. The statistics structure contains a collection of event counts for various packet sent and receive properties. Reading the statistics also clears the current statistic counters, so the values read represent a delta from the last call.The statistics information is copied into the structure pointed to by the pStatistics argument.

Arguments

```
hEMAC       handle to the opened EMAC device

pStatistics  Get the device statistics
```

Return Value
Uint32

- 0 – Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC peripheral instance must be opened before calling this API

Post Condition
1.Statistics are read for various packets sent and received.
2.Reading the statistics also clears the current statistic counters, so the values read represent a delta from the last call.

Modifies
Output parameter *pStatistics* being passed

Example

```
EMAC_Config      ecfg;
EMAC_Statistics  pStatistics;
EMAC_Device      EMACObj;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
...

EMAC_getStatistics(&EMACObj, &pStatistics );
```

8.2.10 EMAC_getStatus

Uint32 EMAC_getStatus	**(**	**Handle**	**hEMAC,**
		EMAC_Status *	**pStatus**
	)		

Description
Called to get the current status of the device. The device status is copied into the supplied data structure.

Arguments

hEMAC	handle to the opened EMAC device
pStatus	Status of the EMAC

Return Value
Uint32

- 0 – Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC peripheral instance must be opened before calling this API.

Post Condition
The current status of the device is copied into the supplied data structure.

Modifies
Output parameter *pStatus* being passed

Example

```
EMAC_Status status;
EMAC_Config  ecfg;
EMAC_Device  EMACObj;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

...


EMAC_getStatus( &EMACObj, &pStatus);
```

8.2.11 EMAC_sendPacket

Uint32 EMAC_sendPacket	**(**	**Handle**	**hEMAC,**
		EMAC_Pkt *	**pPkt**
	)		

Description
Sends a Ethernet data packet out the EMAC device. On a non-error return, the EMAC device takes ownership of the packet. The packet is returned to the application's free pool once it has been transmitted.

The function returns zero on success, or an error code on failure. When an error code is returned, the EMAC device has not taken ownership of the packet.

Arguments

```
hEMAC   handle to the opened EMAC device

pPkt     EMAC packet structure
```

Return Value
`Uint32`

- `0` - Success
- `EMAC_ERROR_INVALID` - A calling parameter is invalid
- `EMAC_ERROR_BADPACKET` - The packet structure is invalid

Pre Condition

EMAC peripheral instance must be opened and get a packet buffer from private queue

Post Condition
Sends a ethernet data packet out the EMAC device and is returned to the application's free pool once it has been transmitted.

Modifies
None

Example

```
#define EMAC_RXFILTER_DIRECT       1
#define EMAC_PKT_FLAGS_SOP         0x80000000u
#define EMAC_PKT_FLAGS_EOP         0x40000000u

EMAC_Config     ecfg;
EMAC_Pkt        *pPkt;
Uint32          size, TxCount = 0;
EMAC_Device     EMACObj;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

//set the receive filter
EMAC_setReceiveFilter( &EMACObj, EMAC_RXFILTER_DIRECT, 0);

//Fill the packet options fields
size = TxCount + 60;
pPkt->Flags      = EMAC_PKT_FLAGS_SOP | EMAC_PKT_FLAGS_EOP;
pPkt->ValidLen   = size;
pPkt->DataOffset = 0;
pPkt->PktChannel = 0;
pPkt->PktLength  = size;
pPkt->PktFrags   = 1;

EMAC_sendPacket( &EMACObj, pPkt );
```

8.2.12 EMAC_RxServiceCheck

Uint32 EMAC_RxServiceCheck (Handle *hEMAC*)

Description
This function should be called every time there is an EMAC device Rx interrupt. It maintains the status the EMAC.
Note that the application has the responsibility for mapping the physical device index to the correct EMAC_serviceCheck() function. If more than one EMAC device is on the same interrupt, the function must be called for each device.
Possible error codes include: EMAC_ERROR_INVALID - A calling parameter is invalid
EMAC_ERROR_MACFATAL - Fatal error in the MAC - Call **EMAC_close()**

Arguments

hEMAC	handle to the opened EMAC device

coreNum	core number

Return Value
Success (0)
EMAC_ERROR_INVALID - A calling parameter is invalid
EMAC_ERROR_MACFATAL - Fatal error in the MAC - Call **EMAC_close()**

Pre Condition
EMAC_open function must be called before calling this API.

Post Condition
None

Modifies
Rx buffers and EMAC Rx head decscriptor pointer and completion pointer registers

Example

```
static CSL_IntcContext              context;
static CSL_IntcEventHandlerRecord   Record[13];
static CSL_IntcObj                  intcEMACRx;
static CSL_IntcHandle               hIntcEMACRx;

//CSL_IntcParam vectId1;
CSL_IntcParam vectId2;

CSL_IntcGlobalEnableState state;

/* Setup the global Interrupt */
context.numEvtEntries = 13;
context.eventhandlerRecord = Record;

/* VectorID for the Event  */
vectId2 = CSL_INTC_VECTID_6;

CSL_intcInit(&context);
/* Enable NMIs  */
CSL_intcGlobalNmiEnable();
/* Enable Global Interrupts  */
CSL_intcGlobalEnable(&state);

/* Opening a handle for EMAC Rx interrupt */

hIntcEMACRx=CSL_intcOpen(&intcEMACRx,
                  CSL_INTC_EVENTID_MACRXINT0,&vectId2,NULL);

/* Hook the ISRs */
CSL_intcHookIsr(vectId2,&HwRxInt);

CSL_intcHwControl(hIntcEMACRx, CSL_INTC_CMD_EVTENABLE, NULL);

/* This function is called when Rx interrupt occurs */
Void HwRxInt (void)
{
  EMAC_Device     EMACObj;
  EMAC_Core       EMACCore;
```

```
    ...

    //Open the EMAC peripheral
    EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

    EMAC_RxServiceCheck(&EMACObj);
}
```

8.2.13 EMAC_setMulticast

Uint32 EMAC_setMulticast	**(**	**Handle**	**hEMAC,**
		Uint32	**AddrCnt,**
		Uint8 *	**pMCastList**
	)		

Description
This function is called to install a list of multicast addresses for use in multicast address filtering. Each time this function is called, any current multicast configuration is discarded in favor of the new list. Thus a set with a list size of zero will remove all multicast addresses from the device.

Note that the multicast list configuration is stateless in that the list of multicast addresses used to build the configuration is not retained. Thus it is impossible to examine a list of currently installed addresses.

The addresses to install are pointed to by pMCastList. The length of this list in bytes is 6 times the value of AddrCnt. When AddrCnt is zero, the pMCastList parameter can be NULL.

The function returns zero on success, or an error code on failure. The multicast list settings are not altered in the event of a failure code.

Arguments

```
hEMAC       handle to the opened EMAC device

AddrCnt   number of addresses to multicast

pMCastList  pointer to the multi cast list
```

Return Value
`Uint32`

- `0` – Success
- `EMAC_ERROR_INVALID` - A calling parameter is invalid

Pre Condition
EMAC peripheral instance must be opened and set multicast filter.

Post Condition
Install a list of multicast addresses for use in multicast address filtering.
A set with a list size of zero will remove all multicast addresses from the device.

Modifies
EMAC registers

Example

```
#define EMAC_RXFILTER_ALLMULTICAST 4
Uint32         AddrCnt;
Uint8        pMCastList;
EMAC_Config  ecfg;
EMAC_Device  EMACObj;
EMAC_Core       EMACCore;

...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACCore);

...

EMAC_setReceiveFilter(&EMACObj, EMAC_RXFILTER_ALLMULTICAST, 0);

EMAC_setMulticast( &EMACObj, AddrCnt, &pMCastList );
```

8.2.14 EMAC_setReceiveFilter

Uint32 EMAC_setReceiveFilter (Handle hEMAC, Uint32 ReceiveFilter, Uint8 masterChannel)

Description
Called to set the packet filter for received packets. The filtering level is inclusive, so BROADCAST would include both BROADCAST and DIRECTED (UNICAST) packets.

Available filtering modes include the following:

EMAC_RXFILTER_NOTHING - Receive nothing
EMAC_RXFILTER_DIRECT - Receive only Unicast to local MAC addr
EMAC_RXFILTER_BROADCAST - Receive direct and Broadcast
EMAC_RXFILTER_MULTICAST - Receive above plus multicast in mcast list
EMAC_RXFILTER_ALLMULTICAST - Receive above plus all multicast
EMAC_RXFILTER_ALL - Receive all packets
Note that if error frames and control frames are desired, reception of these must be specified in the device configuration.

Arguments

hEMAC	handle to the opened EMAC device
ReceiveFilter	Filtering modes
masterChannel	Master core channel to receive the broadcast packets

Return Value
Uint32

- 0 – Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition
EMAC peripheral instance must be opened before calling this API

Post Condition
Sets the packet filter for received packets

Modifies
EMAC registers

Example

```
#define EMAC_RXFILTER_DIRECT       1
EMAC_Config  ecfg;
EMAC_Device  EMACObj;
EMAC_Core       EMACCore;
...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);

...

EMAC_setReceiveFilter(&EMACObj, EMAC_RXFILTER_DIRECT, 0);
```

8.2.15 EMAC_timerTick

Uint32 EMAC_timerTick (Handle hEMAC)

Description
This function should be called for each device in the system on a periodic basis of 100mS (10 times a second). It is used to check the status of the EMAC and MDIO device, and to potentially recover from low Rx buffer conditions.

Strict timing is not required, but the application should make a reasonable attempt to adhere to the 100mS mark. A missed call should not be "made up" by making multiple sequential calls.

A "polling" driver (one that calls EMAC_serviceCheck() in a tight loop), must also adhere to the 100mS timing on this function.

Arguments

hEMAC	handle to the opened EMAC device

Return Value
Uint32

- 0 – Success
- EMAC_ERROR_INVALID - A calling parameter is invalid

Pre Condition

EMAC peripheral instance must be opened

Post Condition
None

Modifies
Re-fill Rx buffer queue if needed and modifies EMAC CONTROL register.

Example

```
EMAC_Config ecfg;
EMAC_Device EMACObj;
EMAC_Core       EMACCore;
...

//Open the EMAC peripheral
EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
...

EMAC_timerTick( &EMACObj );
```

8.2.16 EMAC_txChannelTeardown

static inline int EMAC_txChannelTeardown (*Uint32 val*, *int instNum*)

Description
Tear down selective transmit channel/channels

Arguments

```
val         mask of selective tx channels to be torn down.
InstNum     module instance number
```

Return Value
EMAC_ERROR_INVALID Invalid instance number
0 - Success

Pre Condition
EMAC Tx channels should be in use

Post Condition
Tear down specific tx channels

Modifies
EMAC registers

Example

```
Uint32       val = EMAC_TEARDOWN_CHANNEL(0) |
                   EMAC_TEARDOWN_CHANNEL(1);

             EMAC_txChannelTeardown (val, 0);
```

8.2.17 EMAC_rxChannelTeardown

static inline int EMAC_rxChannelTeardown (*Uint32 val*, *int instNum*)

Description
Tear down selective receive channel/channels

Arguments

```
      val         mask of selective rx channels to be torn down.
      InstNum     module instance number
```

Return Value
EMAC_ERROR_INVALID - Invalid instance number
0 - Success

Pre Condition
EMAC Rx channels should be in use

Post Condition
Tear down specific rx channels

Modifies
EMAC registers

Example

```
Uint32            val = EMAC_TEARDOWN_CHANNEL(0) |
                        EMAC_TEARDOWN_CHANNEL(1);

                  EMAC_rxChannelTeardown (val, 0);
```

8.2.18 EMAC_TxServiceCheck

Uint32 EMAC_TxServiceCheck (Handle *hEMAC*)

Description
This function should be called every time there is an EMAC device Tx interrupt. It maintains the status the EMAC.
Note that the application has the responsibility for mapping the physical device index to the correct EMAC_serviceCheck() function. If more than one EMAC device is on the same interrupt, the function must be called for each device.
Possible error codes include: EMAC_ERROR_INVALID - A calling parameter is invalid
EMAC_ERROR_MACFATAL - Fatal error in the MAC - Call **EMAC_close()**

Arguments

```
      hEMAC             handle to the opened EMAC device
```

Return Value
`Success (0)`
`EMAC_ERROR_INVALID` - A calling parameter is invalid
`EMAC_ERROR_MACFATAL` - Fatal error in the MAC - Call **EMAC_close()**

Pre Condition
EMAC_open function must be called before calling this API.

Post Condition
None

Modifies
Tx buffers and EMAC Tx head decscriptor pointer and completion pointer registers

Example

```
static CSL_IntcContext                context;
static CSL_IntcEventHandlerRecord     Record[13];
static CSL_IntcObj                    intcEMACTx;
static CSL_IntcHandle                 hIntcEMACTx;

   //CSL_IntcParam vectId1;
   CSL_IntcParam vectId2;

   CSL_IntcGlobalEnableState state;

   /* Setup the global Interrupt */
   context.numEvtEntries = 13;
   context.eventhandlerRecord = Record;

   /* VectorID for the Event  */
   vectId2 = CSL_INTC_VECTID_6;

   CSL_intcInit(&context);
   /* Enable NMIs  */
   CSL_intcGlobalNmiEnable();
   /* Enable Global Interrupts  */
   CSL_intcGlobalEnable(&state);

   /* Opening a handle for EMAC Tx interrupt */

      hIntcEMACTx=CSL_intcOpen(&intcEMACTx,
                        CSL_INTC_EVENTID_MACTXINT0,
                       &vectId2,NULL);

   /* Hook the ISR */
   CSL_intcHookIsr(vectId2,&HwTxInt);

   CSL_intcHwControl(hIntcEMACTx, CSL_INTC_CMD_EVTENABLE, NULL);

/* This function is called when Tx interrupt occurs */
Void HwTxInt (void)
{
  EMAC_Device EMACObj;
  EMAC_Core   EMACCore;

  ...

  //Open the EMAC peripheral
  EMAC_open(0, (Handle)0x12345678, &ecfg, &EMACObj, &EMACCore);
  ...

  EMAC_TxServiceCheck(&EMACObj);
}
```

8.3 Data Structures

This section lists Data Structures available in the EMAC module.

8.3.1 EMAC_ChannelInfo

Detailed Description
Transmit/Receive Channel info Structure.
(one receive and up to 8 transmit per core in this example)

Field Documentation
Uint32 EMAC_ChannelInfo::TxChanEnable
Which specific Tx Channels(0-7) to use
if TxChannels = 0 does not allocate the Tx channels for the core
if TxChannels = 1, 2, 4, 8, ... allocates which specific TxChannels to use 0x01: channel 0, 0x02: channel 1, 0x04: channel 2, 0x80: channel 7
User has to take care of allocating a portion from total allocation for the cores
Uint32 EMAC_ChannelInfo::RxChanEnable
Which specific Rx Channels(0-7) to use
if RxChannel = 0 does not allocate the Rx channel for the core
if RxChannel = 1, 2,4,8,... which specific channel to use for the core
0x01: channel 0, 0x02: channel 1, 0x04: channel 2, 0x80: channel 7

8.3.2 EMAC_AddrConfig

Detailed decription
MAC addresses configuration Structure.

Field documentation

Uint8 EMAC_AddrConfig::ChannelNum
Receive Channel number to which the MAC address to be assigned

Uint8 EMAC_AddrConfig:: Addr[6]
MAC address specific to channel

8.3.3 EMAC_Config

Detailed decription
The config structure defines how the EMAC device should operate. It is passed to the device when the device is opened, and remains in effect until the device is closed.

Field documentation

EMAC_Common_Config _EMAC_Config::EMACCommonConfig
Configurations common to all the cores

EMAC_Core_Config _EMAC_Config::EMACCoreConfig
Configurations specific to individual cores

8.3.4 EMAC_Common_Config

Detailed decription
The EMAC_Common_Config structure defines configurations common to all the cors when the EMAC device is operating. It is passed to the device when the device is initialized one time by the master core (EMAC_commonInit()), and remains in effect until the device is de-initialized by the master core (EMAC_commonDeInit()).

Field documentation

Uint8 _EMAC_Common_Config::UseMdio
MDIO Configuation select. User has to pass one (1) if MDIO Configuration is needed, if not should pass zero (0)

Uint32 _EMAC_Common_Config::ModeFlags
Configuration Mode Flags

Uint32 _EMAC_Common_Config::MdioPhyAddr
PHY address (0-31) to be monitored by MDIO, specified by user when MDIO_MODEFLG_SPECPHYADDR is set in MdioModeFlags

Uint32 _EMAC_Common_Config::MdioModeFlags
CSL_MDIO Mode Flags (see MDIO Module)

Uint8 _EMAC_Common_Config::CoreNum
This member is for core selction to do the EMAC configuration i.e user can select the specific core to configure EMAC one time

Uint32 _EMAC_Common_Config::PktMTU
Max physical packet size.

8.3.5 EMAC_Core_Config

Detailed decription
The EMAC_Core_Config structure defines configurations specific to individual cores when the EMAC device is operating. It is passed to the device when the device is initialized for individual cores (EMAC_coreInit()), and remains in effect until the device is de-initialized for the corresponding cores (EMAC_coreDeInit()).

Field documentation

EMAC_Pkt*(* _EMAC_Core_Config::pfcbGetPacket)(Handle hApplication)
Get packet call back

void(* _EMAC_Core_Config::pfcbFreePacket)(Handle hApplication, EMAC_Pkt *pPacket)
Free packet call back

EMAC_Pkt*(* _EMAC_Core_Config::pfcbRxPacket)(Handle hApplication, EMAC_Pkt *pPacket)
Receive packet call back

void(* _EMAC_Core_Config::pfcbStatus)(Handle hApplication)

Get status call back

void(* _EMAC_Core_Config::pfcbStatistics)(Handle hApplication)
Get statistics call back

Uint8 _EMAC_Core_Config::DescBase
This member is for descriptor memory selction to place the EMAC descriptors in CPPI RAM or L2 RAM or DDR memory

EMAC_ChannelInfo _EMAC_Core_Config::ChannelInfo
Tx and Rx Channel information for individual cores to use

Uint8 _EMAC_Core_Config::NumOfMacAddrs
Number of MAC addresses to be assigned for individual cores

EMAC_AddrConfig _EMAC_Core_Config::MacAddr**
Mac Addresses structure

Uint32 _EMAC_Core_Config::RxMaxPktPool
Max Rx packet buffers to get from pool

8.3.6 EMAC_DescCh

Detailed decription
Transmit/Receive Descriptor Channel Structure.

Field documentation

Uint32 EMAC_DescCh::ChannelIndex
Channel index 0-7

Uint32 EMAC_DescCh::DescCount
Current number of descriptors

Uint32 EMAC_DescCh::DescMax
Max number of descriptors (buffs)

PKTQ EMAC_DescCh::DescQueue
Packets queued as descriptor

struct EMAC_Device* _EMAC_DescCh::pd
Pointer to parent structure

EMAC_Desc* EMAC_DescCh::pDescFirst
First descriptor location

EMAC_Desc* EMAC_DescCh::pDescLast
Last descriptor location

EMAC_Desc* EMAC_DescCh::pDescRead
Location to read next descriptor

EMAC_Desc* EMAC_DescCh::pDescWrite

Location to write next descriptor

PKTQ EMAC_DescCh::WaitQueue
Packets waiting for TX descriptor

8.3.7 EMAC_Core

Detailed decription
EMAC Core Instance Structure.

Field documentation

Handle EMAC_Core::hApplication
Calling Application's Handle

EMAC_DescCh EMAC_Core::RxCh[CSL_EMAC_NUMCHANNELS]
Receive channel status

EMAC_DescCh EMAC_Core::TxCh[CSL_EMAC_NUMCHANNELS]
Transmit channel status

8.3.8 EMAC_Device

Detailed decription
EMAC Main Device Instance Structure.

Field documentation

EMAC_Common_Config _EMAC_Device::Config
Original User Configuration common to all cores

EMAC_Core_Config _EMAC_Device::CoreConfig[CSL_EMAC_NUMCORES]
Original User Configuration specific to individaul cores

CSL_EmacRegs* _EMAC_Device::emacRegs
Pointer to the register overlay structure of the EMAC

CSL_EctlRegs* _EMAC_Device::ectlRegs
Pointer to the register overlay structure of the ECTL

CSL_InstNum _EMAC_Device::perNum
Instance of EMAC being referred by this object

Uint32 EMAC_Device::DevMagic
Magic ID for this instance

Uint32 EMAC_Device::FatalError
Fatal Error Code

MDIO_Device EMAC_Device::MdioDev
MDIO device structure

Formatted: French (France)

Uint32 EMAC_Device::MacHash1
Hash value cache

Uint32 EMAC_Device::MacHash2
Hash value cache

Uint32 EMAC_Device::PktMTU
Max physical packet size

Uint32 EMAC_Device::RxFilter
Current RX filter value

EMAC_Statistics EMAC_Device::Stats
Current running statistics

Uint8 _EMAC_Device::InitOnce
EMAC common configuration has been initialized once

Uint8 _EMAC_Device::MacIndexUsed[CSL_EMAC_NUMMACADDRS]
Store core ID which is using each mac index

EMAC_Core* _EMAC_Device:: pEMACCore[CSL_EMAC_NUMCORES]
Array of pointers to EMAC Core Instance for individual cores

8.3.9 EMAC_Pkt

Detailed decription
The packet structure defines the basic unit of memory used to hold data packets for the EMAC device.

Field documentation

Uint32 EMAC_Pkt::AppPrivate
For use by the application

Uint32 EMAC_Pkt::BufferLen
Physical Length of buffer (read only)

Uint32 EMAC_Pkt::DataOffset
Byte offset to valid data

Uint32 EMAC_Pkt::Flags
Packet Flags

Uint8* EMAC_Pkt::pDataBuffer
Pointer to Data Buffer (read only)

Uint32 EMAC_Pkt::PktChannel
Tx/Rx Channel/Priority 0-7 (SOP only)

Uint32 EMAC_Pkt::PktFrags
No of frags in packet (SOP only) frag is EMAC_Pkt record-normally 1

Uint32 EMAC_Pkt::PktLength
Length of Packet (SOP only) (same as ValidLen on single frag Pkt)

struct _EMAC_Pkt* EMAC_Pkt::pNext
Next record

struct _EMAC_Pkt* EMAC_Pkt::pPrev
Previous record

Uint32 EMAC_Pkt::ValidLen
Length of valid data in buffer

8.3.10 EMAC_Statistics

Detailed decription
The statistics structure is used to retrieve the current count of various packet events in the system. These values represent the delta values from the last time the statistics were read.

Field documentation

Uint32 EMAC_Statistics::Frame1024tUp
Total Tx&Rx with Octet Size of >=1024

Uint32 EMAC_Statistics::Frame128t255
Total Tx&Rx with Octet Size of 128 to 255

Uint32 EMAC_Statistics::Frame256t511
Total Tx&Rx with Octet Size of 256 to 511

Uint32 EMAC_Statistics::Frame512t1023
Total Tx&Rx with Octet Size of 512 to 1023

Uint32 EMAC_Statistics::Frame64
Total Tx&Rx with Octet Size of 64

Uint32 EMAC_Statistics::Frame65t127
Total Tx&Rx with Octet Size of 65 to 127

Uint32 EMAC_Statistics::NetOctets
Sum of all Octets Tx or Rx on the Network

Uint32 EMAC_Statistics::RxAlignCodeErrors
Frames Received with Alignment/Code Errors

Uint32 EMAC_Statistics::RxBCastFrames
Good Broadcast Frames Received

Uint32 EMAC_Statistics::RxCRCErrors
Frames Received with CRC Errors

Uint32 EMAC_Statistics::RxDMAOverruns

Total Rx DMA Overruns

Uint32 EMAC_Statistics::RxFiltered
Rx Frames Filtered Based on Address

Uint32 EMAC_Statistics::RxFragments
Rx Frame Fragments Received

Uint32 EMAC_Statistics::RxGoodFrames
Good Frames Received

Uint32 EMAC_Statistics::RxJabber
Jabber Frames Received

Uint32 EMAC_Statistics::RxMCastFrames
Good Multicast Frames Received

Uint32 EMAC_Statistics::RxMOFOverruns
Total Rx Middle of Frame Overruns

Uint32 EMAC_Statistics::RxOctets
Total Received Bytes in Good Frames

Uint32 EMAC_Statistics::RxOversized
Oversized Frames Received

Uint32 EMAC_Statistics::RxPauseFrames
PauseRx Frames Received

Uint32 EMAC_Statistics::RxQOSFiltered
Rx Frames Filtered Based on QoS Filtering

Uint32 EMAC_Statistics::RxSOFOverruns
Total Rx Start of Frame Overruns

Uint32 EMAC_Statistics::RxUndersized
Undersized Frames Received

Uint32 EMAC_Statistics::TxBCastFrames
Good Broadcast Frames Sent

Uint32 EMAC_Statistics::TxCarrierSLoss
Tx Frames Lost Due to Carrier Sense Loss

Uint32 EMAC_Statistics::TxCollision
Total Frames Sent With Collision

Uint32 EMAC_Statistics::TxDeferred
Frames Where Transmission was Deferred

Uint32 EMAC_Statistics::TxExcessiveColl
Tx Frames Lost Due to Excessive Collisions

Uint32 EMAC_Statistics::TxGoodFrames

Good Frames Sent

Uint32 EMAC_Statistics::TxLateColl
Tx Frames Lost Due to a Late Collision

Uint32 EMAC_Statistics::TxMCastFrames
Good Multicast Frames Sent

Uint32 EMAC_Statistics::TxMultiColl
Frames Sent with Multiple Collisions

Uint32 EMAC_Statistics::TxOctets
Total Transmitted Bytes in Good Frames

Uint32 EMAC_Statistics::TxPauseFrames
PauseTx Frames Sent

Uint32 EMAC_Statistics::TxSingleColl
Frames Sent with Exactly One Collision

Uint32 EMAC_Statistics::TxUnderrun
Tx Frames Lost with Tx Underrun Error

8.3.11 EMAC_Status

Detailed decription
EMAC_Status.The status structure contains information about the MAC's run-time status.

Field documentation

Uint32 EMAC_Status::FatalError
Fatal Error when non-zero

Uint32 EMAC_Status::MdioLinkStatus
CSL_MDIO Link status (see MDIO)

Uint32 EMAC_Status::PhyDev
Current PHY device in use (0-31)

Uint32 EMAC_Status::RxPktHeld
Number of packets held for Rx

Uint32 EMAC_Status::TxPktHeld
Number of packets held for Tx

8.3.12 PKTQ

Detailed decription
Packet Queue.

We keep a local packet queue for transmit and receive packets. The queue structure is OS independent.

Field documentation

Uint32 PKTQ::Count
Number of packets in queue

Uint32 EMAC_Pkt* PKTQ::pHead
Pointer to first packet

Uint32 EMAC_Pkt* PKTQ:: pTail
Pointer to last packet

8.4 Macros

#define EMAC_DEVMAGIC **0x0aceface**
Device Magic number

#define CSL_EMAC_NUMSTATS **36**
Number of statistics regs

#define CSL_EMAC_NUMCHANNELS **8**
Number of Tx/Rx channels

#define CSL_EMAC_NUMCORES **6**
Number of cores

#define CSL_EMAC_NUMMACADDRS **32**
Number of cores

#define EMAC_PKT_FLAGS_EOP **0x40000000u**
End of packet

#define EMAC_PKT_FLAGS_SOP **0x80000000u**
Start of packet

#define EMAC_PKT_FLAGS_ALIGNERROR **0x00040000u**
RxErr: Alignment Error

#define EMAC_PKT_FLAGS_CODEERROR **0x00080000u**
RxErr: Code Error

#define EMAC_PKT_FLAGS_CONTROL **0x00200000u**
RxCtl: Control Frame

#define EMAC_PKT_FLAGS_CRCERROR **0x00020000u**
RxErr: Bad CRC

#define EMAC_PKT_FLAGS_FRAGMENT **0x00800000u**
RxErr: Fragment

#define EMAC_PKT_FLAGS_HASCRC **0x04000000u**
RxCrc: PKT has 4byte CRC

#define EMAC_PKT_FLAGS_JABBER **0x02000000u**
RxErr: Jabber

#define EMAC_PKT_FLAGS_NOMATCH **0x00010000u**
RxPrm: No Match

#define EMAC_PKT_FLAGS_OVERRUN **0x00100000u**
RxErr: Overrun

#define EMAC_PKT_FLAGS_OVERSIZE **0x01000000u**
RxErr: Oversize

#define EMAC_PKT_FLAGS_UNDERSIZED 0x00400000u
RxErr: Undersized

#define EMAC_CONFIG_MODEFLG_CHPRIORITY 0x0001
Use Tx channel priority

#define EMAC_CONFIG_MODEFLG_MACLOOPBACK 0x0002
MAC internal loopback

#define EMAC_CONFIG_MODEFLG_PASSCONTROL 0x0020
Pass control frames

#define EMAC_CONFIG_MODEFLG_PASSERROR 0x0010
Pass error frames

#define EMAC_CONFIG_MODEFLG_RXCRC 0x0004
Include CRC in RX frames

#define EMAC_CONFIG_MODEFLG_TXCRC 0x0008
Tx frames include CRC

#define EMAC_CONFIG_MODEFLG_PASSALL 0x00040
Pass all frames

#define EMAC_CONFIG_MODEFLG_RXQOS 0x00080
Enable QOS at receive side

#define EMAC_CONFIG_MODEFLG_RXNOCHAIN 0x00100
Select no buffer chaining

#define EMAC_CONFIG_MODEFLG_RXOFFLENBLOCK 0x00200
Enable offset/length blocking

#define EMAC_CONFIG_MODEFLG_RXOWNERSHIP 0x00400
Use ownership bit as 1

#define EMAC_CONFIG_MODEFLG_RXFIFOFLOWCNTL 0x00800
Enable rx fifo flow control

#define EMAC_CONFIG_MODEFLG_CMDIDLE 0x01000
Enable IDLE command

#define EMAC_CONFIG_MODEFLG_TXSHORTGAPEN 0x02000
Enable tx short gap

#define EMAC_CONFIG_MODEFLG_TXPACE 0x04000
Enable tx pacing

#define EMAC_CONFIG_MODEFLG_TXFLOWCNTL 0x08000
Enable tx flow control

#define EMAC_CONFIG_MODEFLG_RXBUFFERFLOWCNTL 0x10000
Enable rx buffer flow control

#define EMAC_CONFIG_MODEFLG_FULLDUPLEX 0x20000
Set full duplex mode

#define EMAC_CONFIG_MODEFLG_GIGABIT 0x40000
Set gigabit

#define EMAC_CONFIG_MODEFLG_EXTEN 0x80000
Set external enable bit

#define EMAC_CONFIG_MODEFLAG_GIGFORCE 0x100000
Set gigabit force mode

#define EMAC_CONFIG_MODEFLAG_RMIIFULLDUPLEXMODE 0x200000
Set Duplex mode for RMII interface

#define EMAC_CONFIG_MODEFLAG_RMIISPEED100 0x400000
Set operating speed for RMII interface

#define EMAC_DESC_BASE_L2 0x00001
Use L2 as Descriptor memory

#define EMAC_DESC_BASE_CPPI 0x00002
Use CPPI RAM as desriptor memory

#define EMAC_DESC_BASE_DDR 0x00004
Use DDR as descriptor memory

#define EMAC_CONFIG_MODEFLG_GMIIEN 0x800000
Set GMII mode

#define EMAC_TEARDOWN_CHANNEL(x) (1 << x)
Macro to tear down selective Rx/Tx channels

#define EMAC_RXFILTER_ALL 5
Receive filter set to All

#define EMAC_RXFILTER_ALLMULTICAST 4
Receive filter set to All Mcast

#define EMAC_RXFILTER_BROADCAST 2
Receive filter set to Broadcast

#define EMAC_RXFILTER_DIRECT 1
Receive filter set to Direct

#define EMAC_RXFILTER_MULTICAST 3
Receive filter set to Multicast

#define EMAC_RXFILTER_NOTHING 0
Receive filter set to Nothing

#define EMAC_ERROR_ALREADY 1
Operation has already been started

#define EMAC_ERROR_BADPACKET 5
Supplied packet was invalid

#define EMAC_ERROR_DEVICE 3
Device hardware error

#define EMAC_ERROR_INVALID 4
Function or calling parameter is invalid

#define EMAC_ERROR_MACFATAL 6
Fatal Error - EMAC_close() required

#define EMAC_ERROR_NOTREADY 2
Device is not open or not ready

#define ECTL0_REGS ((CSL_EctlRegs *)CSL_ECTL_0_REGS)
EMAC Module instance 0 registers

#define ECTL1_REGS ((CSL_EctlRegs *)CSL_ECTL_1_REGS)
EMAC Module instance 1 registers

8.5 Typedefs

typedef struct _EMAC_Config EMAC_Config
The config structure defines how the EMAC device should operate. It is passed to the device when the device is opened, and remains in effect until the device is closed.

typedef struct _EMAC_Common_Config EMAC_Common_Config
The EMAC_Common_Config structure defines configurations common to all the cors when the EMAC device is operating. It is passed to the device when the device is initialized one time by the master core (EMAC_commonInit()), and remains in effect until the device is de-initialized by the master core (EMAC_commonDeInit()).

typedef struct _EMAC_Core_Config EMAC_Core_Config
The EMAC_Core_Config structure defines configurations specific to individual cores when the EMAC device is operating. It is passed to the device when the device is initialized for individual cores (EMAC_coreInit()), and remains in effect until the device is de-initialized for the corresponding cores (EMAC_coreDeInit()).

typedef struct _EMAC_DescCh EMAC_DescCh
Transmit/Receive Descriptor Channel Structure.
(One receive and up to 8 transmit in this example)

typedef struct _EMAC_Pkt EMAC_Pkt
The packet structure defines the basic unit of memory used to hold data packets for the EMAC device.

typedef struct _EMAC_Statistics EMAC_Statistics
The statistics structure is the used to retrieve the current count of various packet events in the system. These values represent the delta values from the last time the statistics were read.

typedef struct _EMAC_Status EMAC_Status
The status structure contains information about the MAC's run-time status.

typedef struct _EMAC_ChannelInfo EMAC_ChannelInfo
Transmit/Receive Channel info Structure.
(One receive and up to 8 transmit in this example)

typedef struct _EMAC_AddrConfig EMAC_AddrConfig
MAC addresses configuration Structure.

typedef struct _EMAC_Device EMAC_Device
EMAC Main Device Instance Structure.

typedef struct _pktq PKTQ
We keep a local packet queue for transmit and receive packets. The queue structure is OS independent.

Chapter 9
ETB Module

Topics

9.1 Overview

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

The purpose of the embedded trace buffer (ETB) is to provide a buffering mechanism to hold the PC discontinuity trace data before the data leaves the chip. The buffer is a dedicated buffer on chip. In this case, there is a 4KB buffer per core.

9.2 Functions

9.2.1 CSL_etbInit

CSL_Status CSL_etbInit (
CSL_EtbContext * *pContext*)

Description
This is the initialization function for the ETB. This function must be called before calling any other API from this CSL. This function is idem-potent. Currently, the function just returns status CSL_SOK, without doing anything.

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_etbInit(NULL);
...
```

9.2.2 CSL_etbOpen

CSL_EtbHandle CSL_etbOpen

```
(CSL_EtbObj *        pEtbObj,
CSL_InstNum          etbNum,
CSL_EtbParam *       pEtbParam,
CSL_Status *         pStatus
)
```

Description
This function returns the handle to the ETB controller instance. This handle is passed to all other CSL APIs. After successful open, unlocks ETB i.e., by setting the LAR register of ETB in order to enable accesses to any ETB registers.

Arguments

```
pEtbObj          Pointer to etb object.

etbNum           Instance of  DSP ETB to be opened.
```

```
                    There are six instances of the etb
                    available.

    pEtbParam       Module specific parameters.

    pStatus         Status of the function call
```

Return Value
CSL_EtbHandle

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

Pre Condition
CSL_etbInit() must be called successfully in order before calling **CSL_etbOpen()**.

Post Condition
1. The status is returned in the status variable. If status returned is
   - CSL_SOK - Valid etb handle is returned
   - CSL_ESYS_FAIL - The etb instance is invalid
   - CSL_ESYS_INVPARAMS - Invalid parameter

2. ETB object structure is populated

Modifies
1. The status variable
2. ETB object structure

Example

```
    CSL_Status          status;
    CSL_EtbObj          etbObj;
    CSL_EtbHandle       hEtb;
    ...
    hEtb = CSL_etbOpen(&etbObj,
                    CSL_ETB_0,
                    NULL,
                    &status
                    );
    ...
```

9.2.3 CSL_etbclose

CSL_Status CSL_etbClose (CSL_EtbHandle ***hEtb*** **)**

Description
This function closes the specified instance of ETB. After successful close, locks ETB i.e., by resetting the LAR register of ETB in order to disable accesses to any ETB registers.

Arguments

```
    hEtb        Handle to the ETB
```

Return Value
CSL_Status

- CSL_SOK - Close successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both **CSL_etbInit()** and **CSL_etbOpen()** must be called successfully in order before calling **CSL_etbClose()**.

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

Modifies
Obj structure values

Example

```
CSL_EtbHandle           hEtb;
CSL_Status              status;
...

status = CSL_etbClose(hEtb);
...
```

9.2.4 CSL_etbHwControl

CSL_Status CSL_etbHwControl (CSL_EtbHandle *hEtb*,
CSL_EtbControlCmd *cmd*,
void * *arg*
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of ETB.

Arguments

```
hEtb        ETB handle returned by successful 'open'

cmd         The command to this API indicates the action to be
            taken on ETB.
            Control command, refer at CSL_EtbControlCmd for the
            list of commands supported

arg         An optional argument.
            Optional argument as per the control command,
            void * casted
```

Return Value
CSL_Status

- CSL_SOK - Status info return successful.
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVCMD - Invalid command

Pre Condition
Both **CSL_etbInit()** and **CSL_etbOpen()** must be called successfully in order before calling **CSL_etbHwControl()**.
Refer to *CSL_EtbHwControlCmd* for the argument type (*void**) that needs to be passed with the control command

Post Condition
None

Modifies
The hardware registers of ETB.

Example

```
CSL_Status    status;
Uint32        arg;
CSL_EtbHandle hEtb;
...

// Init successfully done
...
// Open successfully done
...

arg = CSL_ETB_TRACEACPEN_ENABLE;
status = CSL_etbHwControl(hEtb,
                          CSL_ETB_CMD_ENA_TRACE_CAPTURE,
                          &arg);
...
```

9.2.5 CSL_etbGetHwStatus

CSL_Status CSL_etbGetHwStatus (CSL_EtbHandle *hEtb*, CSL_EtbHwStatusQuery *query*, void * *response*)

Description
Gets the status of different operations or some setup-parameters of ETB. The status is returned through the third parameter.

Arguments

hEtb	ETB handle returned by successful 'open'
query	The query to this API of ETB which indicates the status to be returned. Query command, refer at CSL_EtbHwStatusQuery for the list of query commands supported
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 - Status info return successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Invalid query command

Pre Condition
Both **CSL_etbInit()** and **CSL_etbOpen()** must be called successfully in order before calling **CSL_etbGetHwStatus()**. Refer to *CSL_EtbHwStatusQuery* for the argument to be passed along with the corresponding query command.

Post Condition
None

Modifies
Third parameter response

Example

```
CSL_EtbHandle     hEtb;
CSL_Status        status;
Uint32            response;
...
status = CSL_etbGetHwStatus(  hEtb,
                              CSL_ETB_QUERY_ACQUISITION_COMPLETE,
                              &response);
...
```

9.2.6 CSL_etbRead

CSL_Status CSL_etbRead (CSL_EtbHandle *hEtb*,
void * rdData
)

Description
This function reads ETB data.

Arguments

hEtb	ETB handle returned by successful 'open'
rdData	read data from the RRD

Return Value
CSL_Status

- CSL_SOK – Read operation successful.
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both **CSL_etbInit()** and **CSL_etbOpen()** must be called successfully in order before calling **CSL_etbRead()**.

Post Condition

None

Modifies
None

Example
```
CSL_Status    status;
Uint32        rdData;
CSL_EtbHandle hEtb;
...

// Init successfully done
...
// Open successfully done
...

status = CSL_etbRead(hEtb, &rdData);
...
```

9.2.7 CSL_etbWrite

CSL_Status CSL_etbWrite (CSL_EtbHandle *hEtb*, void * *wrData*)

Description
This function writes the specified data into ETB data register.

Arguments

hEtb	ETB handle returned by successful 'open'
wrData	write data into the RWD

Return Value
CSL_Status

- CSL_SOK - Success (doesnot verify written data).
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both **CSL_etbInit()** and **CSL_etbOpen()** must be called successfully in order before calling **CSL_etbWrite()**.

Post Condition
Data is written to ETB RAM Write register

Modifies
ETB register.

Example
```
CSL_Status    status;
Uint32        arg;
CSL_EtbHandle hEtb;
...
```

```
// Init successfully done
...
// Open successfully done
...

status = CSL_etbWrite(hEtb, &wrData);
...
```

9.2.8 CSL_etbGetBaseAddress

CSL_Status CSL_etbGetBaseAddress (

CSL_InstNum		***etbNum*,**
CSL_EtbParam *		***pEtbParam*,**
CSL_EtbBaseAddress *	***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_etbOpen()** 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. MMR's go to an alternate location.

Arguments

etbNum	Specifies the instance of the etb to be opened.
pEtbParam	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 etb
- 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_EtbBaseAddress      baseAddress;
...
status = CSL_etbGetBaseAddress(CSL_ETB_0, NULL, &baseAddress);
```

9.3 Data Structures

9.3.1 CSL_EtbBaseAddress

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

Field Documentation

CSL_EtbRegsOvly CSL_EtbBaseAddress::regs
Base-address of the Configuration registers of ETB.

9.3.2 CSL_EtbContext

Detailed Description
ETB specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_EtbContext::contextInfo
Context information of ETB. The below declaration is just a place-holder for future implementation.

9.3.3 CSL_EtbObj

Detailed Description
This structure/object holds the context of the instance of ETB opened using ***CSL_etbOpen()*** function.
Pointer to this object is passed as ETB Handle to all ETB CSL APIs. ***CSL_etbOpen()*** function intializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_EtbObj::perNum
Instance of ETB being referred by this object

CSL_EtbRegsOvly CSL_EtbObj::regs
Pointer to the register overlay structure of the ETB

9.3.4 CSL_EtbParam

Detailed Description
ETB specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation
CSL_BitMask16 CSL_EtbParam::flags
Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

9.4 Enumerations

9.4.1 CSL_EtbControlCmd

This is the set of control commands that are passed to ***CSL_etbHwControl()*** , 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_ETB_CMD_SET_RAM_RD_POINTER	Setup RAM read pointer register **Parameters:***arg – value to set Trace RAM* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_RAM_WR_POINTER	Setup RAM write pointer register **Parameters:***arg – value to set the Trace RAM* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_TRIGG_COUNT	Setup Trigger Counter Register **Parameters:***arg – value to set the trigger counter* **Returns:**CSL_SOK
CSL_ETB_CMD_ENA_TRACE_CAPTURE	Enable Trace Capture **Parameters:***None* **Returns:**CSL_SOK
CSL_ETB_CMD_DIS_TRACE_CAPTURE	Disable Trace Capture **Parameters:***None* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_ENAFORMATTING	Enable Formatting **Parameters:** *arg* (0**-***Disable, 1-Enable)* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_CONT_FORMATTING	Setup Continuous Formatting **Parameters:** *arg* (0**-***Disable, 1-Enable)* **Returns:**CSL_SOK
CSL_ETB_CMD_ENA_FLUSHIN	Enable FLUSHIN **Parameters:***None* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_MANUAL_FLUSH	Setup Manual Flush - FONMAN **Parameters:***None* **Returns:**CSL_SOK
CSL_ETB_CMD_ENA_STOP_FLUSH	Enable stop flush - STOPFI **Parameters:***None* **Returns:**CSL_SOK

CSL_ETB_CMD_SET_ACQCOMP	Setup the value of ACQCOMP **Parameters:***arg (1 –ACQ Complete 0- ACQ not complete)* **Returns:**CSL_SOK
CSL_ETB_CMD_SET_FULL	Setup the value of FULL **Parameters:***arg* (*0-not Full, 1-Full*) **Returns:**CSL_SOK
CSL_ETB_CMD_SET_TRIGACK	Setup the value of TRIGACK **Parameters:***arg* (*0-TrigInAck disable, 1-TrigInACK enable*) **Returns:**CSL_SOK
CSL_ETB_CMD_SET_FLUSHACK	Setup the value of FLUSHACK **Parameters:***arg* (*0-FlushInAck disable*, *1-FlushInACK enable*) **Returns:**CSL_SOK
CSL_ETB_CMD_SET_ATREADY	Setup the value of ATREADY **Parameters:***arg* (*0-Disable, 1-Enable*) **Returns:**CSL_SOK
CSL_ETB_CMD_SET_ATVALID	Setup the value of ATVALID **Parameters:***arg* (*0-Disable, 1-Enable*) **Returns:**CSL_SOK
CSL_ETB_CMD_ENA_STOP_TRIG	Enable stop formatter on trigger - STOPTRIG **Parameters:** *None* **Returns:** CSL_SOK
CSL_ETB_CMD_ENA_FLUSH_ON_TRIG	Enable flush on trigger – FONTRIG **Parameters:** *None* **Returns:** CSL_SOK
CSL_ETB_CMD_ENA_TRIGFL	Enable TRIGFL which indicates a trigger on flush completion – TRIGFL **Parameters:** *None* **Returns:** CSL_SOK
CSL_ETB_CMD_ENA_TRIGEVT	Enable TRIGEVT which indicates a trigger on trigger event – TRIGEVT **Parameters:** *None* **Returns:** CSL_SOK
CSL_ETB_CMD_ENA_TRIGIN	Enable TRIGIN which indicates a trigger on TRIGIN being asserted – TRIGIN **Parameters:** *None* **Returns:** CSL_SOK

9.4.2 CSL_EtbHwStatusQuery

This is the set of query commands to get the status of various operations in ETB
The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_ETB_QUERY_RAM_FULL	Queries the RAM Full (RAM write pointer has wrapped around) **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIG_STAUS	Queries the Trigger bit set when a trigger has been observed **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ACQUISITION_ COMPLETE	Queries the Acquisition complete **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_FORMAT_ PIPELINE	Queries the Formatterpipeline empty, All data stored to RAM **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_FLUSH	Queries the Flush in progress **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_FORMAT_STOP	Queries the Formatter stopped **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ITATBDATA0_ STATUS	Queries the Integration register Status ITATBDATA0 **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ITATBCTR1_ STATUS	Queries the Integration register Status ITATBCTR1 **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ITATBCTR0_ STATUS	Queries the Integration register Status ITATBCTR0 **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_SECURITY_ LEVEL	Queries the Reports security level **Parameters:***(Uint32 *)* **Returns:**CSL_SOK

CSL_ETB_QUERY_DEVICE_ID	Queries the DID - Device ID **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_DEVICE_TYPE	Queries the Device Type Identification Register **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIGIN_VALUE	Queries the value of TRIGIN **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_FLUSHIN_VALUE	Queries the value of FLUSHIN **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_RAM_DEPTH	Queries the value of RAM DEPTH **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_READ_POINTER	Queries the value of READ Pointer **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_WRITE_POINTER	Queries the value of WRITE POINTER **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRACECAP_ STATUS	Queries the value of TRACE Status **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIGGERCOUNT	Queries the value of Trigger Counter **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ ENAFORMATTING	Queries the value of Enable Formatting bit **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_ ENACONTFORMATTING	Queries the value of Enable Continuous Formatting bit **Parameters:***(Uint32 *)* **Returns:**CSL_SOK

CSL_ETB_QUERY_STOPTRIG_ENABLE	Queries the value of STOPTRIG **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_FONTRIG_ENABLE	Queries the value of FONTRIG **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIGFL_ENABLE	Queries the value of TRIGFL **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIGEVT_ENABLE	Queries the value of TRIGEVT **Parameters:***(Uint32 *)* **Returns:**CSL_SOK
CSL_ETB_QUERY_TRIGIN_ENABLE	Queries the value of TRIGIN **Parameters:***(Uint32 *)* **Returns:**CSL_SOK

9.5 Macros

#define CSL_ETB_STS_ACQ_COMPLETE **1**
Acquisition complete

#define CSL_ETB_STS_ACQ_NOTCOMPLETE **0**
Acquisition NOT complete

#define CSL_ETB_UNLOCK_VAL **(0xc5acce55u)**
Value to unlock ETB for register accesses

#define CSL_ETB_TRACEACPEN_ENABLE **1**
Value for trace capture enable

#define CSL_ETB_TRACEACPEN_DISABLE **0**
Value for trace capture disable

9.6 Typedefs

typedef struct CSL_EtbObj CSL_EtbObj
This structure/object holds the context of the instance of ETB opened using ***CSL_etbOpen()*** function

Chapter 10
GPIO Module

Topics

10.1 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, you must first allocate a device using `GPIO_open()`, and then configure the Global Control register to determine the peripheral mode by using the configuration structure.

10.2 Functions

This section lists the functions available in the GPIO module.

10.2.1 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_gpioInit(NULL);
```

10.2.2 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_gpio*Init*() 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

Example

```
CSL_Status          status;
CSL_GpioObj         gpioObj;
CSL_GpioHandle      hGpio;

hGpio = CSL_gpioOpen(&gpioObj, CSL_GPIO, NULL, &status);
```

10.2.3 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);
```

10.2.4 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. But this is a dummy function for this module. It has been kept for future use.

Arguments
```
hGpio          Handle to the GPIO instance

setup          Pointer to hardware setup structure
```

Return Value
Always return CSL_SOK

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
None

Modifies
None

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);
```

10.2.5 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 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_GpioObj         gpioObj;
CSL_GpioConfig      config = CSL_GPIO_CONFIG_DEFAULTS;
CSL_Status          status;

hGpio = CSL_gpioOpen (&gpioObj, CSL_GPIO, NULL, &ststus);

status = CSL_gpioHwSetupRaw (hGpio, &config);
```

10.2.6 CSL_gpioGetHwSetup

CSL_Status CSL_gpioGetHwSetup (CSL_GpioHandle ***hGpio***,
CSL_GpioHwSetup * ***setup***
)

Description
This function gets the current setup of the GPIO. The status is returned through *CSL_GpioHwSetup*. The obtaining of status is the reverse operation of *CSL_gpioHwSetup()* function. *CSL_gpioGetHwSetup* API is reserved for future use.

Arguments

hGpio	Handle to the GPIO instance
setup	Pointer to setup structure which contains the setup information of GPIO.

Return Value
Always return CSL_SOK

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before this function

Post Condition
None

Modifies
None.

Example

```
CSL_GpioHandle      hGpio;
CSL_GpioObj         gpioObj;
CSL_GpioHwSetup     setup;
CSL_Status          status;

hGpio = CSL_gpioOpen (&gpioObj, CSL_GPIO, NULL, &status);

status = CSL_gpioGetHwSetup(hGpio, &setup);
```

10.2.7 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 GPIO pin number
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both *CSL_gpioInit()* and *CSL_gpioOpen()* must be called successfully in order before this function

Post Condition
GPIO registers are configured according to the command passed

Modifies
The hardware registers of GPIO.

Example

```
CSL_GpioHandle          hGpio;
CSL_GpioObj             gpioObj;
CSL_GpioHwControlCmd    cmd = CSL_GPIO_CMD_BANK_INT_ENABLE;
CSL_Status              status;

hGpio = CSL_gpioOpen (&gpioObj, CSL_GPIO, NULL, &status);

status = CSL_gpioHwControl(hGpio, cmd, NULL);
```

10.2.8 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	Place holder 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 this function

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_GpioHandle          hGpio;
CSL_GpioObj             gpioObj;
CSL_GpioHwStatusQuery   query= CSL_GPIO_QUERY_BINTEN_STAT;
void                    response;
CSL_Status              status;

hGpio = CSL_gpioOpen (&gpioObj, CSL_GPIO, NULL, &status);

status = CSL_gpioGetHwStatus(hGpio, query, &response);
```

10.2.9 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 Function call is successful
- 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);
```

10.3 Data Structures

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

10.3.1 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

10.3.2 CSL_GpioConfig

Detailed Description

Config structure of GPIO. This is used to configure GPIO using CSL_HwSetupRaw() 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

10.3.3 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.

10.3.4 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.

10.3.5 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.

10.3.6 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

10.3.7 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

10.3.8 CSL_GpioPinData

Detailed Description
This is used for getting a specific pin status and to set the output value of a pin.

Field Documentation

CSL_GpioPinNum CSL_GpioPinData::pinNum
Pin number for GPIO bank

Int16 CSL_GpioPinData::pinVal
Pin value of the specified pin number

10.4 Enumerations

10.4.1 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

10.4.2 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

10.4.3 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_SET_OUT_BIT	Sets the output state of GPIO pin to logic 1 or logic 0. **Parameters:** *(CSL_GpioPinData**)*

10.4.4 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*)*

10.5 Macros

#define CSL_EGPIO_INVPARAM CSL_EGPIO_FIRST
Value for invalid argument

#define CSL_GPIO_CONFIG_DEFAULTS
Value:

```
{       \
    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.

Chapter 11
HPI Module

Topics

11.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within HPI module. Host Port Interface supports 16-bit and 32-bit.

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.

11.2 Functions

This section lists the functions available in the HPI module.

11.2.1 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);
...
```

11.2.2 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   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 otherwise NULL is returned

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);
```

11.2.3 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);
```

11.2.4 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);
```

11.2.5 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

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_HpiHwControlCmd  cmd = CSL_HPI_CMD_SET_HINT;
void                 arg;
...
status = CSL_hpiHwControl(hHpi, cmd, &arg);
```

11.2.6 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;
void                   response;
...
status = CSL_hpiGetHwStatus(hHpi, query, &response);
```

11.2.7 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_hpiGetHwSetupRaw().

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);
```

11.2.8 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;
...
status = CSL_hpiGetHwSetup(hHpi, &hwSetup);
```

11.2.9 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	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);
```

11.3 Data Structures

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

11.3.1 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

11.3.2 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.

Field Documentation

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

11.3.3 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.

11.3.4 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.

11.3.5 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

11.3.6 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

11.3.7 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

11.4 Enumerations

This section lists the enumerations available in the HPI module.

11.4.1 CSL_HpiHwStatusQuery

enum CSL_HpiHwStatusQuery
Enumeration for hardware status query commands

Enumeration values:

CSL_HPI_QUERY_PID_REV — Query the current value of Peripheral Revision Id.
Parameters:
*(Uint32 *)*

CSL_HPI_QUERY_PID_CLASS — Query the current value of Peripheral Class.
Parameters:
*(Uint32 *)*

CSL_HPI_QUERY_PID_TYPE — Query the current value of Peripheral Type
Parameters:
*(Uint32 *)*

CSL_HPI_QUERY_HRDY — Query the current value of host ready.
Parameters:
*(Uint32 *)*

CSL_HPI_QUERY_FETCH — Query the current value of Host 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 *)*

CSL_HPI_QUERY_DSP_INT — Query the current status of DSP interrupt ie., Host-to-DSP interrupt
Parameters:
*(Uint32 *)*

11.4.2 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)*
CSL_HPI_CMD_RESET_DSPINT	Reset the HPIC Host-to-DSP Interrupt. **Parameters:** *(None)*

11.4.3 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

11.5 Macros

#define CSL_HPI_CONFIG_DEFAULTS

Value:

```
{ \
    CSL_HPI_PWREMU_MGMT_RESETVAL, \
    CSL_HPI_HPIC_RESETVAL, \
    CSL_HPI_HPIAW_RESETVAL, \
    CSL_HPI_HPIAR_RESETVAL \
}
```

Default Values for Config structure

11.6 Typedefs

typedef CSL_HpiObj * CSL_HpiHandle
This data type is used to return the handle to the CSL of the HPI.

Chapter 12
I2C Module

Topics

12.1 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).

12.2 Functions

This section lists the functions available in the I2C module.

12.2.1 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);
...
```

12.2.2 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);
...
```

12.2.3 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);
...
```

12.2.4 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);
...
```

12.2.5 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);
...
```

12.2.6 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. This hold good the commands which takes the parameter as mandatory.

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);
...
```

12.2.7 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
Second parameter

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);
...
```

12.2.8 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);
...
```

12.2.9 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);
...
```

12.2.10 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);
...
```

12.2.11 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);
...
```

12.3 Data Structures

This section lists the data structures available in the I2C module.

12.3.1 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.

12.3.2 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

12.3.3 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.

12.3.4 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.

12.3.5 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

12.3.6 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() andCSL_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

12.3.7 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.

12.4 Enumerations

This section lists the enumerations available in the I2C module.

12.4.1 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_REV	Get the revision level of the I2C. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_CLASS	Get the class of the peripheral. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_TYPE	Get the type of the peripheral **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_SDIR	To get the slave direction. **Parameters:** *(Uint32*)*
CSL_I2C_QUERY_NACKSNT	To get the acknowledgement status. **Parameters:** *(Uint32*)*

12.4.2 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_ALL, 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)*

12.5 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_ALL (0x3Fu)
Clear all status bits

#define CSL_I2C_CLEAR_AL (0x1u)
Clear the Arbitration Lost status bit

#define CSL_I2C_CLEAR_ARDY (0x4u)
Clear the Register access ready status bit

#define CSL_I2C_CLEAR_NACK (0x2u)
Clear the No acknowledge status bit

#define CSL_I2C_CLEAR_RRDY (0x8 u)
Clear the Receive ready status bit

#define CSL_I2C_CLEAR_SCD (0x20u)
Clear the Stop Condition Detect status bit

#define CSL_I2C_CLEAR_XRDY (0x10u)
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

12.6 Typedefs

typedef CSL_I2cObj * CSL_I2cHandle
Handle to the I2C object Handle is used in all accesses to the device parameters.

Chapter 13
INTC MODULE

Topics

13.1 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.

13.2 Functions

This section lists the functions available in the INTC module.

13.2.1 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);
}
...
```

13.2.2 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);
...
```

13.2.3 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);
...
```

13.2.4 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 - Successful 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)
   {
     ...
   }
```

13.2.5 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() {
    ...
}
```

13.2.6 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);
...
```

13.2.7 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);
...
```

13.2.8 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);
...
```

13.2.9 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);
...
```

13.2.10 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);
...
```

13.2.11 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();
...
```

13.2.12 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();
...
```

13.2.13 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();
...
```

13.2.14 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);
...
```

13.2.15 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);
...
```

13.2.16 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);
...
```

13.2.17 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);
...
```

13.2.18 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() must be called before calling 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);
...
```

13.2.19 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);
...
```

13.2.20 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);
...
```

13.2.21 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);
...
```

13.2.22 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);
...
```

13.2.23 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);
...
```

13.2.24 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);
...
```

13.2.25 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);
...
```

13.2.26 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);
...
```

13.2.27 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_intcCombinedEventClear (eventId, clearMask);
...
```

13.2.28 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_intcCombinedEventGet(eventId);
```

13.2.29 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);
        ...
```

13.2.30 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);
        ...
```

13.2.31 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);
...
```

13.2.32 CSL_intcInterruptDropEnable

void CSL_intcInterruptDropEnable (CSL_BitMask32 *dropMask*)

Description
This API enables interrupts for 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 );
...
```

13.2.33 CSL_intcInterruptDropDisable

void CSL_intcInterruptDropDisable (CSL_BitMask32 *dropMask*)

Description
This API disables interrupts for drop detection.

Arguments

```
dropMask – Vectorid Mask
```

Return Value
None

Pre Condition
None

Post Condition
None

Modifies
INTC hardware registers

Example:

```
CSL_BitMask32      dropMask;
...
CSL_ intcInterruptDropDisable(dropMask);
...
```

13.2.34 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);
...
```

13.2.35 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);
...
```

13.2.36 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);
...
```

13.2.37 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);
...
```

13.2.38 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_ intcInterruptRestore(vectId, restoreVal);
...
```

13.2.39 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);
...
```

13.2.40 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);
...
```

13.2.41 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_ intcQueryInterruptStatus(vectId);
...
```

13.2.42 CSL_intcExcepEnable

CSL_IntcEventEnableState CSL_intcExcepEnable(CSL_IntcEventId *eventId*)

Description
This API enables the specified exception event.

Arguments

eventide	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);
...
```

13.2.43 CSL_intcExcepDisable

CSL_IntcEventEnableState CSL_intcExcepDisable(CSL_IntcEventId *eventId*)

Description
This API disables the specified exception event..

Arguments

eventide	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);
...
```

13.2.44 CSL_intcExcepRestore

void CSL_intcExcepRestore (CSL_IntcEventId *eventId*, Uint32 *restoreVal*)

Description
This API restores the specified exception event.

Arguments

eventide	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);
...
```

13.2.45 CSL_intcExcepClear

void CSL_intcExcepClear (CSL_IntcEventId *eventId*)

Description
This API enables the specified exception event.

Arguments

eventide	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);
...
```

13.3 Data Structures

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

13.3.1 CSL_IntcObj

Detailed description
The interrupt handle object. This object is used referenced by the handle to identify the event.

Field Documentation

CSL_IntcEventId CSL_IntcObj::eventId
The event-id

CSL_IntcVectId CSL_IntcObj::vectId
The vector-id

13.3.2 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

13.3.3 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

13.3.4 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

13.4 Enumerations

This section lists the enumerations available in the INTC module.

13.4.1 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.

13.4.2 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. **Parameters:** *CSL_IntcEnableState*
CSL_INTC_CMD_EVTSET	Sets the event manually. **Parameters:** *None*
CSL_INTC_CMD_EVTCLEAR	Clears the event (if pending). **Parameters:** *None*
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. **Parameters:** *CSL_IntcEnableState*

13.4.3 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*

13.4.4 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_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
CSL_INTC_EXCEP_0TO31	Symbol for EXPMASK[0]. **Parameters:** *BitMask* for EXPMASK0

13.4.5 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*

13.5 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

Chapter 14
Mdio Module

Topics

14.1 Overview

The Management Data Input/Output (MDIO) module implements the 802.3 serial management interface to interrogate and controls up to 32 Ethernet PHY(s) connected to the device, using a shared two-wire bus. Application software uses the MDIO module to configure the auto-negotiation parameters of each PHY attached to the EMAC, retrieve the negotiation results, and configure required parameters in the EMAC module for correct operation. The module is designed to allow almost transparent operation of the MDIO interface, with very little maintenance from the core processor. There is an optional MDIO configuration parameter provided in EMAC configuration structure. If EMAC CSL functions are used (like EMAC_open and EMAC_timerTick) MDIO functions should not be used by application.

14.2 Functions

This section lists Functions available in the MDIO module.

14.2.1 MDIO_initPHY

Uint32 MDIO_initPHY (Handle hMDIO,
volatile Uint32 phyAddr
)

Description
Force a switch to the specified PHY, and start the negotiation process.

Arguments

```
hMDIO       handle to the opened MDIO instance

phyAddr     Physical address
```

Return Value
1.Returns 1 if the PHY selection completed OK,
2.else 0

Pre Condition
MDIO module must be reset.

Post Condition
Initializes the specific PHY device.

Modifies
PHY registers

Example

```
#define MDIO_MODEFLG_FD1000        0x0020
#define MDIO_MODEFLG_LOOPBACK      0x0040

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

//Open the MDIO module
MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_initPHY( &mdioDev, mdioPhyAddr );
```

14.2.2 MDIO_open

Uint32 MDIO_open (int instNum,
Uint32 mdioModeFlags,

```
                    Uint32          phyAddr,
                    Handle          hMDIO
                )
```

Description
Opens the MDIO peripheral and start searching for a PHY device. It is assumed that the MDIO module is reset prior to calling this function.

Arguments

```
instNum      Instance number when user is specifying a PHY address
             for monitoring, i.e., MDIO_MODEFLG_SPECPHYADDR mode flag
             is set. It is hard-coded as 0 when
             MDIO_MODEFLG_SPECPHYADDR is not set.

mdioModeFlag Mode flags

phyAddr      PHY address user specifies for monitoring. It is
             bypassed when MDIO_MODEFLG_SPECPHYADDR mode flag is not
             set.

hMDIO        Handle to the MDIO device which needs to be initialized.
```

Return Value
Success (0)
MDIO_ERROR_INVALID - A calling parameter is invalid

Pre Condition
The MDIO module must be reset prior to calling this function.

Post Condition
Opens the MDIO peripheral and start searching for a PHY device.

Modifies
MDIO configuration registers

Example

```
#define MDIO_MODEFLG_FD1000         0x0020
#define MDIO_MODEFLG_LOOPBACK       0x0040

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);
```

14.2.3 MDIO_close

void MDIO_close (Handle hMDIO)

Description
Close the MDIO peripheral and disable further operation

Arguments

```
hMDIO       handle to the opened MDIO instance
```

Return Value
None

Pre Condition
MDIO module must be reset and opened before calling this function.

Post Condition
MDIO module is closed. No further operations are possible.

Modifies
PHY and MDIO registers

Example

```
#define MDIO_MODEFLG_FD1000         0x0020
#define MDIO_MODEFLG_LOOPBACK       0x0040

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_close( &mdioDev );
```

14.2.4 MDIO_getStatus

void MDIO_getStatus (Handle **hMDIO,** Uint32 * **pPhy,** Uint32 * **pLinkStatus** **)**

Description
Called to get the status of the MDIO/PHY

Arguments

```
hMDIO             handle to the opened MDIO instance

pPhy              pointer to the physical address

pLinkStatus       pointer to the link status
```

Return Value
None

Pre Condition
MDIO module must be reset and opened before calling this API

Post Condition
MDIO status is returned through the parameters passed in this API.

Modifies
Output parameters *pPhy* and *pLinkStatus* being passed

Example

```
#define MDIO_MODEFLG_FD1000         0x0020
#define MDIO_MODEFLG_LOOPBACK       0x0040

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_getStatus( &mdioDev,&pPhy,&pLinkStatus );
```

14.2.5 MDIO_phyRegRead

Uint32 MDIO_phyRegRead (volatile Uint32 phyIdx, volatile Uint32 phyReg, Uint16 * pData)

Description
Raw data read of a PHY register.

Arguments

```
phyIdx      Physical Index

phyReg      Physical register

pData       Data read
```

Return Value
1. Returns 1 if the PHY ACK'd the read,
2. else 0

Pre Condition
MDIO module must be reset and opened. PHY device must be initialized.

Post Condition
Raw data is read from a PHY register.

Modifies
Output parameter *pData* being passed.

Example

```
#define MDIO_MODEFLG_FD1000        0x0020
#define MDIO_MODEFLG_LOOPBACK      0x0040

volatile Uint32 phyIdx = PHYREG_CONTROL;
volatile Uint32 phyReg;
Uint16 pData;

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

//Open the MDIO module
MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_initPHY( &mdioDev, mdioPhyAddr );

MDIO_phyRegRead( phyIdx, ((MDIO_Device *)&mdioDev)->phyReg,
                 pData );
```

14.2.6 MDIO_phyRegWrite

Uint32 MDIO_phyRegWrite (volatile Uint32 phyIdx,
volatile Uint32 phyReg,
Uint16 pData,
)

Description
Raw data write of a PHY register.

Arguments

phyIdx	physical index
phyReg	Physical register
pData	Data written

Return Value
1. Returns 1 if the PHY ACK'd the write,
2. else 0

Pre Condition
MDIO module must be reset and opened. PHY device must be initialized.

Post Condition
Raw data is written to a PHY register.

Modifies
None

Example

```
#define MDIO_MODEFLG_FD1000       0x0020
#define MDIO_MODEFLG_LOOPBACK     0x0040
#define PHYREG_SHADOW_EXTLOOPBACK 0x8400

volatile Uint32 phyIdx = PHYREG_CONTROL;
volatile Uint32 phyReg;
Uint16 pData = PHYREG_SHADOW_EXTLOOPBACK;

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

//Open the MDIO module
MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_initPHY( &mdioDev, mdioPhyAddr );

MDIO_phyRegWrite( phyIdx, ((MDIO_Device *)&mdioDev)->phyReg,
                  pData );
```

14.2.7 MDIO_timerTick

Uint32 MDIO_timerTick (Handle hMDIO)

Description
Called to signify that approx 100mS have elapsed

Arguments

```
hMDIO       Handle to the opened MDIO instance
```

Return Value
MDIO event code (see MDIO Events).

Pre Condition
MDIO module must be reset and opened before calling this API.

Post Condition
Gets approximately 100mS delay

Modifies
PHY and MDIO registers

Example

```
#define MDIO_MODEFLG_FD1000       0x0020
#define MDIO_MODEFLG_LOOPBACK     0x0040

MDIO_Device    mdioDev;
int            instNum = 0;
Uint32         mdioModeFlags = MDIO_MODEFLG_FD1000 |
                               MDIO_MODEFLG_LOOPBACK;
Uint32         mdioPhyAddr = 0;

//Open the MDIO module
MDIO_open(instNum, mdioModeFlags, mdioPhyAddr, &mdioDev);

MDIO_timerTick( &mdioDev );
```

14.3 Data Structures

This section lists Data Structures available in the MDIO module.

14.3.1 MDIO_Device

Detailed decription
This is the MDIO object that contains the MDIO device object characteristics.

Field documentation

Uint32 MDIO_Device::LinkStatus
Link State PHYREG_STATUS_LINKSTATUS

Uint32 MDIO_Device::ModeFlags
User specified configuration flags

Uint32 MDIO_Device::PendingStatus
Pending Link Status

Uint32 MDIO_Device::phyAddr
Current (or next) PHY addr (0-31)

Uint32 MDIO_Device::phyState
PHY State

Uint32 MDIO_Device::phyStateTicks
Ticks elapsed in this PHY state

Uint32 MDIO_Device:: regId
The set of MDIO User Access and PHY Select Registers used to monitor the PHY: 0 - USERACCESS0 and USERPHYSEL0; 1 - USERACCESS1 and USERPHYSEL1

14.4 Enumerations

enum Interface
Interface. MDIO Interface enum

14.5 Macros

#define DEV_REGS ((CSL_DevRegs *) CSL_DEV_REGS)
Base address of device registers

#define MDIO_REGS ((CSL_MdioRegs *) CSL_MDIO_0_REGS)
Base address of MDIO registers

#define VBUSCLK 165
Standard defines/assumptions for MDIO interface

#define MDIO_MODEFLG_AUTONEG 0x0001
Use Autonegotiate

#define MDIO_MODEFLG_EXTLOOPBACK 0x0100
Use external PHY Loopback, with plug

#define MDIO_MODEFLG_FD10 0x0004
Use 10Mb/s Full Duplex

#define MDIO_MODEFLG_FD100 0x0010
Use 100Mb/s Full Duplex

#define MDIO_MODEFLG_FD1000 0x0020
Use 1000Mb/s Full Duplex

#define MDIO_MODEFLG_HD10 0x0002
Use 10Mb/s Half Duplex

#define MDIO_MODEFLG_HD100 0x0008
Use 100Mb/s Half Duplex

#define MDIO_MODEFLG_LOOPBACK 0x0040
Use PHY Loopback

#define MDIO_MODEFLG_EXTLOOPBACK 0x0100
Use external PHY Loopback, with plug

#define MDIO_MODEFLG_SPECPHYADDR 0x0200
Monitor PHY address which is specified by user

#define MDIO_MODEFLG_NWAYACTIVE 0x0080
NWAY currently active

#define MDIO_LINKSTATUS_FD10 2
Link Status: FD10

#define MDIO_LINKSTATUS_FD100 4
Link Status: FD100

#define MDIO_LINKSTATUS_FD1000 5
Link Status: FD1000

#define MDIO_LINKSTATUS_HD10 1
Link Status: HD10

#define MDIO_LINKSTATUS_HD100 3
Link Status: HD100

#define MDIO_LINKSTATUS_NOLINK 0
Link Status: No Link

#define MDIO_EVENT_LINKDOWN 1
Link down event

#define MDIO_EVENT_LINKUP 2
Link (or re-link) event

#define MDIO_EVENT_NOCHANGE 0
No change from previous status

#define MDIO_EVENT_PHYERROR 3
No PHY connected

#define MDIO_ERROR_INVALID 1
Function or calling parameter is invalid

#define PHYREG_CONTROL 0
Control register

#define PHYREG_CONTROL_AUTONEGEN (1<<12)
Auto Negate Enable bit

#define PHYREG_CONTROL_AUTORESTART (1<<9)
Set Auto restart bit

**#define PHYREG_CONTROL_DUPLEXFULL (1<<8) **
Set Full Duplex bit

#define PHYREG_CONTROL_ISOLATE (1<<10)
Set Isolate bit

#define PHYREG_CONTROL_LOOPBACK (1<<14)
Set Loop back bit

#define PHYREG_CONTROL_POWERDOWN (1<<11)
Set Power Down bit

#define PHYREG_CONTROL_RESET (1<<15)
Set Reset bit

#define PHYREG_CONTROL_SPEEDLSB (1<<13)
Set Speed LSB bit

#define PHYREG_CONTROL_SPEEDMSB (1<<6)
Set Speed MSB bit

#define PHYREG_STATUS 1 /< Status register */**

Status register

#define PHYREG_STATUS_AUTOCAPABLE (1<<3)
Set Auto Capable bit

#define PHYREG_STATUS_AUTOCOMPLETE (1<<5)
Set Auto complete bit

#define PHYREG_STATUS_EXTENDED (1<<0)
Set Extended bit

#define PHYREG_STATUS_EXTSTATUS (1<<8)
Set External Status bit

#define PHYREG_STATUS_FD10 (1<<12)
Set FD10 bit

#define PHYREG_STATUS_FD100 (1<<14)
Set FD100 bit

#define PHYREG_STATUS_HD10 (1<<11)
Set HD10bit

#define PHYREG_STATUS_HD100 (1<<13)
Set HD100 bit

#define PHYREG_STATUS_JABBER (1<<1)
Set Jabber bit

#define PHYREG_STATUS_LINKSTATUS (1<<2)
Set Link status bit

#define PHYREG_STATUS_NOPREAMBLE (1<<6)
Set No preamble bit

#define PHYREG_STATUS_REMOTEFAULT (1<<4)
Set Remote default bit

#define PHYREG_ID1 2
Physical ID 1 register

#define PHYREG_ID2 3
Physical ID 1 register

#define PHYREG_ADVERTISE 4
Physical Advertise reg

#define PHYREG_ADVERTISE_FAULT (1<<13)
Set Fault bit

#define PHYREG_ADVERTISE_FD10 (1<<6)
Set FD10 bit

#define PHYREG_ADVERTISE_FD100 (1<<8)

Set FD100 bit

#define PHYREG_ADVERTISE_HD10 (1<<5)
Set HD10 bit

#define PHYREG_ADVERTISE_HD100 (1<<7)
Set HD100 bit

#define PHYREG_ADVERTISE_MSG (1)
Set Message bit

#define PHYREG_ADVERTISE_MSGMASK (0x1F)
Set Message mask bit

#define PHYREG_ADVERTISE_NEXTPAGE (1<<15)
Set next page bit

#define PHYREG_ADVERTISE_PAUSE (1<<10)
Set Pause bit

#define PHYREG_1000CONTROL 9
Physical 1000 Ctrl reg

#define PHYREG_1000STATUS 0xA
Phy 1000 Status reg

#define PHYREG_ADVERTISE_FD1000 (1<<9)
Advertise FD1000 bit

#define PHYREG_EXTSTATUS 0x0F
Physical Ext status reg

#define PHYREG_EXTSTATUS_FD1000 (1<<13)
Ext Status FD1000 bit

#define PHYREG_PARTNER 5
Physical Partner reg

#define PHYREG_PARTNER_ACK (1<<14)
Set Acknowledge bit

#define PHYREG_PARTNER_FAULT (1<<13)
Set Fault bit

#define PHYREG_PARTNER_FD10 (1<<6)
Set FD10 bit

#define PHYREG_PARTNER_FD100 (1<<8)
Set FD100 bit

#define PHYREG_PARTNER_FD1000 (1<<11)
Partner FD1000 bit

#define PHYREG_PARTNER_HD10 (1<<5)

Set HD10 bit

#define PHYREG_PARTNER_HD100 (1<<7)
Set HD100 bit

#define PHYREG_PARTNER_MSGMASK (0x1F)
Set Message mask bit

#define PHYREG_PARTNER_NEXTPAGE (1<<15)
Set next page bit

#define PHYREG_PARTNER_PAUSE (1<<10)
Set Pause bit

#define PHYREG_ACCESS 0x1C
Physical Access

#define PHYREG_ACCESS_COPPER 0xFC00
Access Copper

#define PHYREG_SHADOW 0x18
Physical shadow reg

#define PHYREG_SHADOW_EXTLOOPBACK 0x8400
Shadow Ext Loopback bit

#define PHYREG_SHADOW_INBAND 0xF1C7
Shadow In band bit

#define PHYREG_SHADOW_RGMIIMODE 0xF080
Shadow RGMII mode bit

#define PHYSTATE_LINKED 5
MDIO Linked

#define PHYSTATE_LINKWAIT 4
MDIO Wait for link

#define PHYSTATE_MDIOINIT 0
MDIO Initialization state

#define PHYSTATE_NWAYSTART 2
MDIO N Way start

#define PHYSTATE_NWAYWAIT 3
MDIO N Way wait

#define PHYSTATE_RESET 1
MDIO Reset State

```
#define PHYREG_read(regadr, phyadr)                                    \
        MDIO_REGS->USERACCESS0 =                                       \
        CSL_FMK(MDIO_USERACCESS0_GO,1u)              |  \
        CSL_FMK(MDIO_USERACCESS0_REGADR,regadr) |  \
```

CSL_FMK(MDIO_USERACCESS0_PHYADR,phyadr)

This macro reads from the PHY register through the USERACESS register provided in MDIO module

```
#define PHYREG_write(regadr, phyadr, data)                          \
        MDIO_REGS->USERACCESS0 =                                    \
        CSL_FMK(MDIO_USERACCESS0_GO,1u)             |               \
        CSL_FMK(MDIO_USERACCESS0_WRITE,1)           |       \
        CSL_FMK(MDIO_USERACCESS0_REGADR,regadr) |       \
        CSL_FMK(MDIO_USERACCESS0_PHYADR,phyadr) |       \
        CSL_FMK(MDIO_USERACCESS0_DATA, data)
```

This macro writes to the PHY register through the USERACESS register provided in MDIO module

```
#define PHYREG_wait()                               \
    while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) )
```

Waits for GO bit to set

```
#define PHYREG_waitResults( results )
{                                   \
    while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) );     \
    results = CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_DATA);
}
```

Waits for GO bit to set and reads data from the PHY register

```
#define PHYREG_waitResultsAck( results, ack )
{                                           \
    while( CSL_FEXT(MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_GO) );    \
    results = CSL_FEXT( MDIO_REGS->USERACCESS0,MDIO_USERACCESS0_DATA ); \
    ack = CSL_FEXT( MDIO_REGS->USERACCESS0, MDIO_USERACCESS0_ACK);
}
```

Waits for GO bit to set, reads data from the PHY register and checks for ACK

14.6 Typedefs

typedef void* Handle
Void pointer type defined for MDIO

Chapter 15
PDMA Module

Topics

15.1 Overview

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

The PDMA module is a DMA controller element of a distributed DMA topology. The PDMA module is a VBUS/VBUSP compliant module that enables automatic data transfers between other VBUSP compliant modules. PDMA module used to provide data transfers between a DSP subsystem and a peripheral. The interface of the PDMA module and the DSP subsystem may be through a switch. There will be one PDMA module per peripheral.

The PDMA Interface Manager (PIM) is an interface bridge to the PDMA. The purpose of the PDMA Interface Manager is to assure the integrity of PDMA programming in a multi-core environment. PIM interfaces to the config bus.

PDMA as master of both UTOPIA and VBUSM SCR, which then transfers data from UTOPIA to any other slave peripheral.

15.2 Functions

This section lists the Functions available in the PDMA module.

15.2.1 CSL_pdmaInit

CSL_Status CSL_pdmaInit (CSL_PdmaContext * pContext)

Description
This is the initialization function for the PDMA 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
This function should be called before using any of the CSL PDMA APIs

Post Condition
The CSL for PDMA is initialized

Modifies
None

Example

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

15.2.2 CSL_pdmaOpen

```
CSL_PdmaHandle CSL_pdmaOpen   ( CSL_PdmaObj      *pPdmaObj,
                                CSL_PdmaNum      pdmaNum,
                                CSL_PdmaParam    pPdmaParam,
                                CSL_Status       *pStatus
                              )
```

Description
This function returns the handle to the PDMA instance. The open call sets up the data structures for the particular instance of PDMA. The handle returned by this call is input argument for rest of the PDMA CSL APIs.

Arguments

pPdmaObj	Pointer to the PDMA instance object.
pdmaNum	Instance of the PDMA to be opened.
pPdmaParam	Pointer to module specific parameters.
pStatus	Pointer to the variable that holds the status of the open call

Return Value
CSL_PdmaHandle

- Valid PDMA handle will be returned if status value is equal to CSL_SOK otherwise NULL is returned

Pre Condition
The PDMA must be successfully initialized via CSL_pdmaInit() before calling this function. Memory for the CSL_PdmaObj must be allocated outside this call. This object must be retained while using this peripheral instance.

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

CSL_SOK - Valid PDMA handle is returned
CSL_ESYS_FAIL - The PDMA instance is invalid
CSL_ESYS_INVPARAMS - Invalid parameter

2. PDMA object structure is populated.

Modifies
1. The status variable
2. object structure

Example:

```
CSL_PdmaHandle    hPdma;
CSL_PdmaObj       pdmaObj;
CSL_PdmaHwSetup   pdmaSetup;
CSL_Status        status;
...
hPdma = CSL_pdmaOpen(&pdmaObj, CSL_PDMA, NULL, &status);
...
```

15.2.3 CSL_pdmaClose

CSL_Status CSL_pdmaClose (CSL_PdmaHandle hPdma)

Description
Unreserves the PDMA identified by the handle passed.

Arguments

hPdma	Handle to the PDMA

Return Value
CSL_Status

- CSL_SOK - Close Successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both CSL_pdmaInit() and CSL_pdmaOpen() must be called successfully in order before calling CSL_pdmaClose().

Post Condition
PDMA CSL APIs can not be called until the PDMA CSL is reopened again using CSL_pdmaOpen().

Modifies
PDMA Handle

Example

```
CSL_Status        status;
CSL_pdmaHandle    hPdma;
...
status = CSL_pdmaClose(hPdma);
...
```

15.2.4 CSL_pdmaHwControl

CSL_Status CSL_pdmaHwControl (CSL_PdmaHandle hPdma, CSL_PdmaControlCmd ctrlCmd, Uint32 eventGrp)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of PDMA.

Arguments

```
hPdma       PDMA handle returned by successful 'open'

ctrlCmd     The command to this API indicates the action to be
            taken PDMA. Control command, refer @a
            CSL_PdmaControlCmd for the list of commands supported

eventGrp    Hardware setup for the specified event group
```

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_pdmaInit() and CSL_pdmaOpen() must be called successfully in order before calling CSL_pdmaHwControl().

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

Modifies
The hardware registers of PDMA.

Example

```
CSL_Status        status;
CSL_BitMask16     ctrlMask;
CSL_PdmaHandle    hPdma;
...
// Init successfully done
...
// Open successfully done
...
status = CSL_pdmaHwControl(hPdma, CSL_PDMA_QUERY_GLOBAL_STATUS,
                           EVENT_GROUP_ONE);
```

15.2.5 CSL_pdmaHwSetup

CSL_Status CSL_pdmaHwSetup	**(**	**CSL_PdmaHandle**	**hPdma,**
		CSL_PdmaHwSetup	***setup,**
		Uint32	**eventGroup**
	)		

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_PdmaHwSetup.

Arguments

hPdma	PDMA handle returned by successful 'open'
setup	Pointer to the intialized setup structure
eventGroup	Hardware setup for the specified event group

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_pdmaInit() and CSL_pdmaOpen() must be called successfully in order before calling CSL_pdmaHwSetup().

Post Condition
The registers of the specified PDMA instance will be setup according to value passed.

Modifies
PDMA registers

Example

```
CSL_PdmaHandle    hPdma;
CSL_PdmaHwSetup setup = CSL_PDMA_HWSETUP_DEFAULTS;
...
// Init Successfully done
...
// Open Successfully done
...
CSL_pdmaHwSetup(hPdma, &setup);
...
```

15.2.6 CSL_pdmaChHwControl

CSL_Status CSL_pdmaChHwControl (CSL_PdmaHandle **hPdma,**
CSL_PdmaControlCmd **ctrlCmd,**
Uint32 chnum
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of PDMA channel.

Arguments

hPdma	PDMA handle returned by successful 'open'
ctrlCmd	The command to this API indicates the action to be taken on PDMA. Control command, refer @a CSL_PdmaControlCmd for the list of commands supported
chnum	PDMA channel number

Return Value
CSL_Status

- CSL_SOK - Channel Hw command Successful
- CSL_ESYS_BADHANDLE - Handle passed is invalid
- CSL_ESYS_INVCMD - Invalid command

Pre Condition
Both CSL_pdmaInit() and CSL_pdmaOpen() must be called successfully in order before calling CSL_ pdmaChHwControl ().

Post Condition
The registers of the specified PDMA instance will be setup according to value passed.

Modifies
PDMA registers

Example

```
CSL_Status        status;
CSL_PdmaHandle    hPdma;
CSL_PdmaControlCmd    ctrlCmd = CSL_PDMA_CMD_CHANNEL_ENABLE;
...

// Init successfully done
...
// Open successfully done
...

status = CSL_pdmaChHwControl(hPdma, ctrlCmd, 0);
...
```

15.2.7 CSL_pdmaChHwSetup

CSL_Status CSL_pdmaChHwSetup (CSL_PdmaHandle hPdma, CSL_PdmaChHwSetup * chSetup, Uint32 chNum)

Description
This function initializes the pdma channel..

Arguments

hPdma	PDMA handle returned by successful 'open'
chSetup	Pointer to the intialized channel setup structure
chNum	Hardware Pdma channel number

Return Value
CSL_Status

- CSL_SOK – channel Hwsetup Successful
- CSL_ESYS_BADHANDLE - Handle passed is invalid
- CSL_ESYS_INVPARAMS - The param passed is invalid

Pre Condition
Both CSL_pdmaInit() and CSL_pdmaOpen() must be called successfully in order before calling CSL_pdmaChHwSetup().

Post Condition
The registers of the specified PDMA instance will be setup according to value passed.

Modifies
PDMA registers

Example

```
CSL_PdmaHandle hPdma;
CSL_pdmaChHwSetup  txChSetup;
CSL_pdmaChHwSetup  rxChSetup;
...
// Initialize txChSetup & rxChSetup.  For this initialization,
// please refer the utopia2 example.
...

// Init Successfully done
...
// Open Successfully done
...
CSL_pdmaChHwSetup(hPdma, &chSetup);
...
```

15.2.8 CSL_pdmaGetHwSetup

CSL_Status CSL_pdmaGetHwSetup (CSL_PdmaHandle ***hPdma***,
CSL_PdmaHwSetup * ***Setup***,
Uint32 ***eventGroup***
)

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

Arguments

hPdma	Handle to the PDMA
setup	Pointer to the hardware setup structure
eventGroup	Hardware setup for the specified event group

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_pdmaInit()* and *CSL_pdmaOpen()* 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_PdmaHandle     hPdma;
CSL_Status         status;
CSL_PdmaHwSetup    *setup
...
// pdma is initialized and opened successfully.
...
status = CSL_pdmaGetHwSetup (hPdma, setup);
...
```

15.2.9 CSL_pdmaGetHwStatus

CSL_Status CSL_pdmaGetHwStatus (CSL_PdmaHandle *hPdma*,
CSL_PdmaHwStatusQuery *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_PdmaHwStatusQuery*. User should allocate memory for the said data type and pass its pointer as an unadorned void* argument to the status query call.

Arguments

```
hPdma      Handle to the PDMA instance

query     The query to this API of PDMA 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_pdmaInit()* and *CSL_pdmaOpen()* must be called successfully in order before calling *CSL_pdmaGetHwStatus()*.

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_PdmaHandle    hPdma;
void              reponse;
...
status = CSL_pdmaGetHwStatus (hPdma,
```

```
                                    CSL_PDMA_QUERY_STATISTIC_REGS,
                                    &response);
    ...
```

15.2.10 CSL_ pdmaGetBaseAddress

CSL_Status CSL_pdmaGetBaseAddress (CSL_InstNum *pdmaNum*,
CSL_PdmaParam * *pPdmaParam*,
CSL_PdmaBaseAddress * *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_pdmaOpen() 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

PdmaNum	Specifies the instance of the pdma to be opened.
pPdmaParam	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 PDMA.
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Inavlid 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_PdmaBaseAddress   baseAddress;
    ...
    status = CSL_pdmaGetBaseAddress(CSL_PDMA, NULL,
                                    &baseAddress);
```

15.3 Data Structures

This section lists the Data Structures available in the PDMA module.

15.3.1 CSL_BufSize

Detailed Description
The structure contains the buffer size.

Field Documentation

volatile Uint16 CSL_BufSize::pbuf
Peripheral Buffer Size bit

volatile Uint16 CSL_BufSize::sbuf
Switch Buffer Size bit

15.3.2 CSL_ChannelContext

Detailed Description
The structure contains the channel context information.

Field Documentation

volatile Uint32 CSL_ChannelContext:: SAR
Source Address Register

volatile Uint32 CSL_ChannelContext:: PAR
Peripheral Address Register

volatile CSL_BufSize CSL_ChannelContext:: cnt_buff
Transfer Count/Size register

volatile CSL_ChanTferCtl CSL_ChannelContext:: chTctl
Transfer control register

15.3.3 CSL_ChanTferCtl

Detailed Description
The structure contains the channel transfer control.

Field Documentation

volatile Uint16 CSL_ChanTferCtl::bcz
Block Count Equal Zero bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::ff
FIFO Full Flag bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::ie
Interrupt Enable bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::imod

Interrupt Mode bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::psiz
Peripheral element size bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::rdptr
Read Pointer bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::sint
Subsystem Interrupt bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::ssiz
Switch element size bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::strt
start bit of PDMA Transfer Control Register

volatile Uint16 CSL_ChanTferCtl::wrptr
Write Pointer bit of PDMA Transfer Control Register

15.3.4 CSL_ChHwSetup

Detailed Description
The structure contains the channel setup.

Field Documentation

CSL_ChannelContext *CSL_ChHwSetup::chContext
Pdma channel context hardware setup

15.3.5 CSL_PdmaContext

Detailed Description
PDMA specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_PdmaContext::contextInfo
Context information of Pdma. The below declaration is just a place-holder for future implementation.

15.3.6 CSL_PdmaGblSetup

Detailed Description
The structure contains the pdma global setup.

Field Documentation

volatile Uint32 CSL_PdmaGblSetup::free
Free Running Mode Control bit of PDMA Global Control and Status Register

volatile Uint32 CSL_PdmaGblSetup::soft

Soft Emulation Halt Control bit of PDMA Global Control and Status Register

volatile Uint32 CSL_PdmaGblSetup::strt
Start Control bit of PDMA Global Control and Status Register

15.3.7 CSL_PdmaHwSetup

Detailed Description
The structure contains the PDMA hardware setup.

Field Documentation

CSL_PdmaGblSetup* CSL_PdmaHwSetup::gbl
Pdma global setup

CSL_PdmaPeriCtrlSetup* CSL_PdmaHwSetup::pctl
Pdma peripheral control setup

15.3.8 CSL_PdmaObj

Detailed Description
This structure/object holds the context of the instance of PDMA opened using CSL_pdmaOpen() function. Pointer to this object is passed as PDMA Handle to all PDMA CSL APIs. CSL_pdmaOpen() function initializes this structure based on the parameters passed.

Field Documentation

CSL_InstNum CSL_PdmaObj::perNum
Instance of pdma being referred by this object

CSL_PdmaRegs CSL_PdmaObj::regs
Pointer to the register overlay structure of the hpi

15.3.9 CSL_PdmaPeriCtrlSetup

Detailed Description
The structure contains the pdma peripheral control setup.

Field Documentation

volatile Uint16 CSL_PdmaPeriCtrlSetup::abu
Auto-buffering Mode

volatile Uint16 CSL_PdmaPeriCtrlSetup::bend
Pdma big endian peripheral control setup

volatile Uint16 CSL_PdmaPeriCtrlSetup::clrs
Clear Statistics Register

volatile Uint16 CSL_PdmaPeriCtrlSetup::dir
Direction of transfer

volatile Uint16 CSL_PdmaPeriCtrlSetup::pblen

Peripheral burst length

volatile Uint16 CSL_PdmaPeriCtrlSetup::pe
Peripheral Event Control

volatile Uint16 CSL_PdmaPeriCtrlSetup::pmod
Peripheral address modification mode

volatile Uint16 CSL_PdmaPeriCtrlSetup::pri
priority

volatile Uint16 CSL_PdmaPeriCtrlSetup::sblen
Switch burst length

volatile Uint16 CSL_PdmaPeriCtrlSetup::sfrm
Super-Frame Sync Select

volatile Uint16 CSL_PdmaPeriCtrlSetup::smod
Switch address modification

volatile Uint16 CSL_PdmaPeriCtrlSetup::sync
Peripheral Synchronization Start

15.3.10 CSL_PdmaPerId

Detailed Description
The structure contains the pdma peripheral id.

Field Documentation

Uint16 CSL_PdmaPerId::cid
Peripheral Class ID

Uint16 CSL_PdmaPerId::prev
Peripheral Revision Number

Uint16 CSL_PdmaPerId::tid
Peripheral Type ID

15.4 Enumerations

This section lists the Enumerations available in the PDMA module.

15.4.1 CSL_PdmaAddrMod

enum CSL_PdmaAddrMod
Pdma address mode select enumeration.

Enumeration values

CSL_PDMA_ADDR_INCR	Pdma address increment
CSL_PDMA_ADDR_CONST	Pdma address constant

15.4.2 CSL_PdmaAutoBufCtrl

enum CSL_PdmaAutoBufCtrl
Pdma auto buffer control select enumeration.

Enumeration values

CSL_PDMA_ABU_DIS	Pdma Auto-buffering Mode Disable
CSL_PDMA_SYNC_ENB	Pdma sync enable

15.4.3 CSL_PdmaChXferCtrl

enum CSL_PdmaChXferCtrl
Pdma channel select enumeration.

Enumeration values

CSL_PDMA_CH_XFER_DIS	Pdma channel XFER disable
CSL_PDMA_CH_XFER_ENB	Pdma channel XFER enable

15.4.4 CSL_PdmaControlCmd

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

Enumeration values

CSL_PDMA_CMD_GLOBAL_ENABLE	Pdma global enable
CSL_PDMA_CMD_GLOBAL_DISABLE	Pdma global disable
CSL_PDMA_CMD_GLOBAL_STATUS	Pdma global status
CSL_PDMA_CMD_GET_STATISTICS	Pdma get global statistics
CSL_PDMA_CMD_CLR_STATISTICS	Pdma clear global statistics
CSL_PDMA_CMD_GET_CHANNEL	Pdma get channel
CSL_PDMA_CMD_CLOSE_CHANNEL	Pdma close channel
CSL_PDMA_CMD_CHANNEL_ENABLE	Pdma channel enable
CSL_PDMA_CMD_CHANNEL_DISABLE	Pdma channel disable

CSL_PDMA_CMD_INTERRUPT_ENABLE	Pdma interrupt enable
CSL_PDMA_CMD_INTERRUPT_DISABLE	Pdma interrupt disable
CSL_PDMA_CMD_PERI_CTL_SYNC_EN	Pdma peripheral control sync enable
CSL_PDMA_CMD_PERI_CTL_SYNC_DIS	Pdma peripheral control sync disable
CSL_PDMA_CMD_PERI_CTL_SUPERFS_EN	Pdma super frame sync operation enable
CSL_PDMA_CMD_PERI_CTL_SUPERFS_DIS	Pdma super frame sync operation disable

15.4.5 CSL_PdmaDataElementSize

enum CSL_PdmaDataElementSize
Pdma data element size select enumeration.

Enumeration values

CSL_PDMA_DATA_ELEMENT_8BIT	Pdma 8 bit data element
CSL_PDMA_DATA_ELEMENT_16BIT	Pdma 16 bit data element
CSL_PDMA_DATA_ELEMENT_32BIT	Pdma 32 bit data element

15.4.6 CSL_PdmaDirCtrl

enum CSL_PdmaDirCtrl
Pdma direction control select enumeration.

Enumeration values

CSL_PDMA_SW_PER	Pdma direction control from Sw to Per
CSL_PDMA_PER_SW	Pdma direction control from Per to Sw

15.4.7 CSL_PdmaEmu

enum CSL_PdmaEmu
Pdma emulation select enumeration.

Enumeration values

CSL_PDMA_EMU_STOP	Pdma emulation stop
CSL_PDMA_EMU_FREERUN	free running mode

15.4.8 CSL_PdmaEnable

enum CSL_PdmaEnable
Pdma global control select enumeration

Enumeration values

CSL_PDMA_GLOBAL_DIS	Pdma global disable
CSL_PDMA_GLOBLE_ENB	Pdma global enable

15.4.9 CSL_PdmaEndian

enum CSL_PdmaEndian
Pdma Endianness select enumeration

Enumeration values

CSL_PDMA_ENDIAN_LITTLE	Pdma little endian mode
CSL_PDMA_ENDIAN_BIG	Pdma big endian mode

15.4.10 CSL_PdmaHwStatusQuery

enum CSL_PdmaHwStatusQuery
Enumeration for queries passed to CSL_pdmaGetHwStatus().
This is used to get the status of different operations.

Enumeration values

CSL_PDMA_QUERY_PID	Pdma query of PID
CSL_PDMA_QUERY_STATISTIC_REGS	Pdma query of register statistic
CSL_PDMA_QUERY_PERIPHERAL_CTRL_REGS	Pdma query of peripheral control register
CSL_PDMA_QUERY_MEMORY_CTRL_REGS	Pdma query of memory control register
CSL_PDMA_QUERY_ALL_REGS	Pdma query of all registers
CSL_PDMA_QUERY_GLOBAL_STATUS	Pdma query of global status
CSL_PDMA_QUERY_CHANNEL_INFO	Pdma query of channel information

15.4.11 CSL_PdmaIntCtrl

enum CSL_PdmaIntCtrl
Pdma interrupt control select enumeration.

Enumeration values

CSL_PDMA_INT_DIS	Pdma interrupt disable
CSL_PDMA_INT_ENB	Pdma interrupt enable

15.4.12 CSL_PdmaIntMod

enum CSL_PdmaIntMod
Pdma interrupt mode select enumeration.

Enumeration values

CSL_PDMA_INT_HALFBUF	Pdma interrupt mode half buffer
CSL_PDMA_INT_FULLBUF	Pdma interrupt mode full buffer
CSL_PDMA_INT_BLKEND	Pdma interrupt mode block end

15.4.13 CSL_PdmaPerSyncStart

enum CSL_PdmaPerSyncStart
Pdma peripheral sync select enumeration.

Enumeration values

CSL_PDMA_NOSYNC_XFER	Pdma no sync
CSL_PDMA_SYNC_XFER	Pdma sync

15.4.14 CSL_PdmaPriority

enum CSL_PdmaPriority
Pdma priority select enumeration

Enumeration values

CSL_PDMA_PRIORITY_0	Pdma priority 0
CSL_PDMA_PRIORITY_1	Pdma priority 1
CSL_PDMA_PRIORITY_2	Pdma priority 2
CSL_PDMA_PRIORITY_3	Pdma priority 3

15.4.15 CSL_PdmaSframeSync

enum CSL_PdmaSframeSync
Pdma frame sync select enumeration.

Enumeration values

CSL_PDMA_SFRMSYNC_RST	Pdma Super-Frame Sync reset
CSL_PDMA_SFRMSYNC_SET	Pdma Super-Frame Sync set

15.4.16 CSL_PdmaSubsysIntSel

enum CSL_PdmaSubsysIntSel
Pdma sub system interrupt select enumeration.

Enumeration values

CSL_PDMA_INT_CPU0	Pdma interrupt for CPU 0
CSL_PDMA_INT_CPU1	Pdma interrupt for CPU 1
CSL_PDMA_INT_CPU2	Pdma interrupt for CPU 2
CSL_PDMA_INT_CPU3	Pdma interrupt for CPU 3
CSL_PDMA_INT_CPU4	Pdma interrupt for CPU 4
CSL_PDMA_INT_CPU5	Pdma interrupt for CPU 5
CSL_PDMA_INT_CPU6	Pdma interrupt for CPU 6
CSL_PDMA_INT_CPU7	Pdma interrupt for CPU 7

15.5 Macros

#define EVENT_GROUP_EIGHT 7
PDMA Event group 8

#define EVENT_GROUP_FIVE 4
PDMA Event group 5

#define EVENT_GROUP_FOUR 3
PDMA Event group 4

#define EVENT_GROUP_ONE 0
PDMA Event group 1

#define EVENT_GROUP_SEVEN 6
PDMA Event group 7

#define EVENT_GROUP_SIX 5
PDMA Event group 6

#define EVENT_GROUP_THREE 2
PDMA Event group 3

#define EVENT_GROUP_TWO 1
PDMA Event group 2

15.6 Typedefs

typedef CSL_PdmaPerId CSL_PdmaPerId
The structure contains the pdma peripheral id.

typedef CSL_PdmaHwSetup CSL_PdmaHwSetup
The structure contains the PDMA hardware setup.

typedef CSL_ChHwSetup CSL_ChHwSetup
The structure contains the channel setup.

typedef CSL_PdmaObj CSL_PdmaObj
This structure/object holds the context of the instance of PDMA opened using CSL_pdmaOpen() function. Pointer to this object is passed as PDMA Handle to all PDMA CSL APIs. CSL_pdmaOpen() function initializes this structure based on the parameters passed.

typedef CSL_PdmaObj * CSL_PdmaHandle
This data type is used to return the handle to the CSL of PDMA.

Chapter 16
PLLC Module

Topics

16.1 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).
The PLL1 controller features a software-programmable PLL multiplier controller (PLLM) and ten dividers (D1 to D10). The PLL1 controller uses the device input clock CLKIN1 to generate a system reference clock (SYSREFCLK) and ten system clocks (SYSCLK1 to SYSCLK10). Only the divider ratio bits of divider D10 (for SYSCLK10) are programmable through the PLL controller divider registers PLLDIV10 and the remaining clock dividers are fixed.

The secondary PLL controller generates interface clocks for the Ethernet media access controller (EMAC) .The PLL2 controller features a PLL multiplier controller and six dividers (D1-D6). The PLL multiplier is fixed and the divider D1-D6 can be programmed through registers PLLDIV1-PLLDIV6.

The third PLL3 controller generates interface clocks for DDR.

16.2 Functions

This section lists the functions available in the PLLC module.

16.2.1 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_pllcInit(NULL);
```

16.2.2 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  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
None

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);
```

16.2.3 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
None

Post Condition
None

Modifies
The peripheral object structure.

Example

```
CSL_PllcHandle   hPllc;
CSL_Status       status;
...

status = CSL_pllcClose(hPllc);
```

16.2.4 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 this function is called.

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;
CSL_Status       status;
...

hPllc = CSL_pllcOpen(&pllcObj, CSL_PLLC_1, NULL, &status);
...

hwSetup.divEnable  = (CSL_BitMask32) 0x00000001;
hwSetup.pllM       = (Uint32)        0x00000001;

status = CSL_pllcHwSetup(hPllc, &hwSetup);
```

16.2.5 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, PLLDIVn, PLLCTL_PLLEN bit must be in BYPASS mode

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_PllcHwControlCmd   cmd = CSL_PLLC_CMD_SET_PLLM;
Uint32   arg = 0x00000002;
...

status = CSL_pllcHwControl (hPllc, cmd, &arg);
```

16.2.6 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
None

Post Condition
None

Modifies
The input argument "response" is modified.

Example

```
CSL_PllcHandle          hPllc;
CSL_PllcHwStatusQuery   query = CSL_PLLC_QUERY_STATUS;
CSL_BitMask32           response;
...

status = CSL_pllcGetHwStatus (hPllc, query, &response);
```

16.2.7 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
None

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;
CSL_Status           status;
...

status = CSL_pllcHwSetupRaw (hPllc, &config);
```

16.2.8 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
None

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

Modifies
hwSetup variable

Example

```
CSL_PllcHandle   hPllc;
CSL_PllcHwSetup  hwSetup;

status = CSL_pllcGetHwSetup(hPllc, &hwSetup);
```

16.2.9 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	Module specific parameters
pBaseAddress	Pointer to base address structure containing base address details.

Return Value
CSL_Status

- CSL_SOK - Open call is successful
- 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);
```

16.3 Data Structures

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

16.3.1 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

16.3.2 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

Volatile Uint32 CSL_PllcConfig::PLLCTL
PLL Control register.

volatile Uint32 CSL_PllcConfig::PLLM
PLL Multiplier Control register

volatile Uint32 CSL_PllcConfig::PLLDIV1
PLL Controller Divider 1 register. This should be configured only for instance 2

volatile Uint32 CSL_PllcConfig::PLLDIV2
PLL Controller Divider 2 register. This should be configured only for instance 2

volatile Uint32 CSL_PllcConfig::PLLDIV3
PLL Controller Divider 3 register. This should be configured only for instance 2

Volatile Uint32 CSL_PllcConfig::PLLCMD
PLL Controller Command register

Volatile Uint32 CSL_PllcConfig::ALNCTL
PLL Controller Clock Align Control register

volatile Uint32 CSL_PllcConfig::PLLDIV4
PLL Controller Divider 2 register. This should be configured only for instance 2

volatile Uint32 CSL_PllcConfig::PLLDIV5
PLL Controller Divider 5 register. This should be configured only for instance 2

volatile Uint32 CSL_PllcConfig::PLLDIV6
PLL Controller Divider 6 register. This should be configured only for instance 2

volatile Uint32 CSL_PllcConfig::PLLDIV7
PLL Controller Divider 6 register. This should be configured only for instance 1

volatile Uint32 CSL_PllcConfig::PLLDIV8
PLL Controller Divider 6 register. This should be configured only for instance 1

volatile Uint32 CSL_PllcConfig::PLLDIV9
PLL Controller Divider 6 register. This should be configured only for instance 1

volatile Uint32 CSL_PllcConfig::PLLDIV10
PLL Controller Divider 10 register. This should be configured only for instance 1

16.3.3 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.

16.3.4 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.

Uint32 CSL_PllcHwSetup::pllM
PLL Multiplier. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllDiv1
PLL Divider 1. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv2
PLL Divider 2. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv3
PLL Divider 3. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv4
PLL Divider 4. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv5
PLL Divider 5. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv6
PLL Divider 6. This is valid only for PLLC instance 2

Uint32 CSL_PllcHwSetup::pllDiv7
PLL Divider 6. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllDiv8
PLL Divider 6. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllDiv9
PLL Divider 6. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup::pllDiv10
PLL Divider 10. This is valid only for PLLC instance 1

Uint32 CSL_PllcHwSetup:: phaseAlign
Phase Align Control

void* CSL_PllcHwSetup::extendSetup
Setup that can be used for future implementation

16.3.5 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.

16.3.6 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

16.3.7 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

16.3.8 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

16.4 Enumerations

This section lists the enumerations available in the PLLC module.

16.4.1 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

16.4.2 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. **Parameters:** *CSL_BitMask32* **See also:** CSL_PLLC_CTRL_DEFINE
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* **See also:** CSL_PllcOscDivCtrl

CSL_PLLC_CMD_SET_PHASEALIGN — By setting the ALN bits, the selected SYSCLKx (specified by input bitmask) will always be phase aligned to other clocks also selected in this register

Parameters:
CSL_BitMask32

16.4.3 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_PID — Queries PLL Control Peripheral ID.
Parameters:
(Uint32)*

CSL_PLLC_QUERY_STATUS — Queries PLL Controller Status.
Parameters:
(CSL_BitMask32)*
See also:
CSL_PLLC_STATUS_DEFINE

CSL_PLLC_QUERY_DIVRATIO_CHANGE — Queries PLLDIV Modified Status.
Parameters:
(CSL_BitMask32)*
See also:
CSL_PLLC_DCHANGESTAT_DEFINE

CSL_PLLC_QUERY_SYSCLKSTAT — Queries PLL SYSCLK Status.
Parameters:
(CSL_BitMask32)*
See also:
CSL_PLLC_SYSCLKSTAT_DEFINE

CSL_PLLC_QUERY_RESETSTAT — Queries Reset Type Status.
Parameters:
(CSL_BitMask32)*

See also:
CSL_PLLC_RESETSTAT_DEFINE

CSL_PLLC_QUERY_EFUSEERR — Queries Fuse Farm Error Status.
Parameters:
(Uint32)*

16.5 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

#define CSL_PLLC_STATUS_STABLE CSL_FMKT (PLLC_PLLSTAT_STABLE, YES)
Set when OSCIN/CLKIN is assumed to be stable

PLLC Divider Ratio Modified Status

#define CSL_PLLC_DCHANGESTAT_SYS1 CSL_FMKT (PLLC_DCHANGE_SYS1, YES)
SYSCLK13 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS2 CSL_FMKT (PLLC_DCHANGE_SYS2, YES)
SYSCLK14 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS3 CSL_FMKT (PLLC_DCHANGE_SYS3, YES)
SYSCLK15 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS4 CSL_FMKT (PLLC_DCHANGE_SYS4, YES)
SYSCLK16 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS5 CSL_FMKT (PLLC_DCHANGE_SYS5, YES)
SYSCLK17 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS6 CSL_FMKT (PLLC_DCHANGE_SYS6, YES)
SYSCLK18 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS7 CSL_FMKT (PLLC_DCHANGE_SYS7, YES)
SYSCLK7 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS8 CSL_FMKT (PLLC_DCHANGE_SYS8, YES)
SYSCLK8 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS9 CSL_FMKT (PLLC_DCHANGE_SYS9, YES)
SYSCLK9 divide ratio is modified

#define CSL_PLLC_DCHANGESTAT_SYS10 CSL_FMKT (PLLC_DCHANGE_SYS10, YES)
SYSCLK10 divide ratio is modified

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_XWRST CSL_FMKT (PLLC_RSTYPE_XWRST, YES)
External Warm Reset

PLLC Control Mask

#define CSL_PLLC_CTRL_ALIGN_PHASE (CSL_FMKT (PLLC_PLLCMD_GOSET, SET)<< 16)
A write of 1 to this bit signifies that the new divide ratios in PLLDIV[1:n] are taken into account at the nearest possible rising edge to phase align the clocks. The actual SYSCLKx to be aligned are selected in register ALNCTL

#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

PLLC Align Control

#define CSL_PLLC_ALIGNCTL_SYSCLK10 CSL_FMKT (PLLC_ALNCTL_ALN10, YES)
SYSCLK10 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK9 CSL_FMKT (PLLC_ALNCTL_ALN9, YES)
SYSCLK9 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK8 CSL_FMKT (PLLC_ALNCTL_ALN8, YES)
SYSCLK8 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK7 CSL_FMKT (PLLC_ALNCTL_ALN7, YES)
SYSCLK7 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK6 CSL_FMKT (PLLC_ALNCTL_ALN6, YES)
SYSCLK18 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK5 CSL_FMKT (PLLC_ALNCTL_ALN5, YES)
SYSCLK17 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK4 CSL_FMKT (PLLC_ALNCTL_ALN4, YES)
SYSCLK16 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK3 CSL_FMKT (PLLC_ALNCTL_ALN3, YES)
SYSCLK15 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK2 CSL_FMKT (PLLC_ALNCTL_ALN2, YES)
SYSCLK14 needs to be aligned with other clocks selected in this register

#define CSL_PLLC_ALIGNCTL_SYSCLK1 CSL_FMKT (PLLC_ALNCTL_ALN1, YES)
SYSCLK13 needs to be aligned with other clocks selected in this register

PLLC Divider Enable

#define CSL_PLLC_DIVEN_PLLDIV1 (1 << 3)
Enable divider D1 for SYSCLK13

#define CSL_PLLC_DIVEN_PLLDIV2 (1 << 4)
Enable divider D1 for SYSCLK14

#define CSL_PLLC_DIVEN_PLLDIV3 (1 << 5)
Enable divider D1 for SYSCLK15

#define CSL_PLLC_DIVEN_PLLDIV4 (1 << 6)
Enable divider D1 for SYSCLK16

#define CSL_PLLC_DIVEN_PLLDIV5 (1 << 7)
Enable divider D1 for SYSCLK17

#define CSL_PLLC_DIVEN_PLLDIV6 (1 << 8)
Enable divider D1 for SYSCLK18

#define CSL_PLLC_DIVEN_PLLDIV7 (1 << 9)
Enable divider D1 for SYSCLK7

#define CSL_PLLC_DIVEN_PLLDIV8 (1 << 10)
Enable divider D1 for SYSCLK8

#define CSL_PLLC_DIVEN_PLLDIV9 (1 << 11)
Enable divider D1 for SYSCLK9

#define CSL_PLLC_DIVEN_PLLDIV10 (1 << 12)
Enable divider D1 for SYSCLK10

Divider Select for SYSCLKs

#define CSL_PLLC_DIVSEL_PLLDIV1 (1)
Divider D1 for SYSCLK13

#define CSL_PLLC_DIVSEL_PLLDIV2 (2)
Divider D2 for SYSCL14

#define CSL_PLLC_DIVSEL_PLLDIV3 (3)
Divider D3 for SYSCLK15

#define CSL_PLLC_DIVSEL_PLLDIV4 (4)
Divider D4 for SYSCLK16

#define CSL_PLLC_DIVSEL_PLLDIV5 (5)
Divider D5 for SYSCLK17

#define CSL_PLLC_DIVSEL_PLLDIV6 (6)
Divider D6 for SYSCLK18

#define CSL_PLLC_DIVSEL_PLLDIV7 (7)
Divider D6 for SYSCLK7

#define CSL_PLLC_DIVSEL_PLLDIV8 (8)
Divider D6 for SYSCLK8

#define CSL_PLLC_DIVSEL_PLLDIV9 (9)
Divider D6 for SYSCLK9

#define CSL_PLLC_DIVSEL_PLLDIV10 (10)
Divider D10 for SYSCLK10

#define CSL_PLLC_HWSETUP_DEFAULTS

Value:

```
{ \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    0,                                  \
    NULL                                \

}
```

Default hardware setup parameters.

PLLC Default Config Structure

#define CSL_PLLC_CONFIG_DEFAULTS
Value:

```
{   \
   CSL_PLLC_PLLCTL_RESETVAL,       \
   CSL_PLLC_PLLM_RESETVAL,         \
   CSL_PLLC_PLLDIV1_RESETVAL,      \
   CSL_PLLC_PLLDIV2_RESETVAL,      \
   CSL_PLLC_PLLDIV3_RESETVAL,      \
   CSL_PLLC_PLLCMD_RESETVAL,       \
   CSL_PLLC_PLLDIV4_RESETVAL,      \
   CSL_PLLC_PLLDIV5_RESETVAL,      \
   CSL_PLLC_PLLDIV6_RESETVAL,      \
   CSL_PLLC_PLLDIV7_RESETVAL,      \
   CSL_PLLC_PLLDIV8_RESETVAL,      \
```

```
    CSL_PLLC_PLLDIV9_RESETVAL,      \
    CSL_PLLC_PLLDIV10_RESETVAL      \
}
```

Default values for config structure

Chapter 17
PSC Module

Topics

17.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within PSC module. The PSC manages clocks to the various modules. The PSC allows software to gracefully gate clocks to any unused modules and to reset these modules during device operation.

The PSC consists of one global power/sleep controller (GPSC) and many local, per-module, power/sleep controllers (LPSCs). Each module's reset and clocks are controlled by the LPSC.

Tomahawk comprises several power domains to enable minimizing power dissipation for unused logic on the device. The GPSC manages each of the power domains. Additionally, clock gating to each of the logic blocks is manages by the LPSCs of each module

.

17.2 Functions

This section lists the functions available in the PSC module.

17.2.1 CSL_pscInit

CSL_Status CSL_pscInit (CSL_PscContext * *pContext*)

Description
This is the initialization function for the PSC 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 PSC 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 power/sleep controller is initialized.

Modifies
None

Example

```
CSL_pscInit(NULL);
```

17.2.2 CSL_pscOpen

CSL_PscHandle CSL_pscOpen (CSL_PscObj * *pPscObj*,
CSL_InstNum *pscNum*,
CSL_PscParam * *pPscParam*,
CSL_Status * *pStatus*)

Description
This function populates the peripheral data object for the PSC instance and returns a handle to the instance. The open call sets up the data structures for the particular instance of PSC 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 PSC CSL APIs.

Arguments

pPscObj	Pointer to PSC object.
pscNum	Instance of PSC CSL to be opened.
pPscParam	Module specific parameters
pStatus	Status of the function call

Return Value
CSL_PscHandle

- Valid PSC handle will be returned if status value is equal to CSL_SOK.
- Returns invalid parameters if status value is equal to CSL_ESYS_INVPARAMS.

Pre Condition
The PSC must be successfully initialized via CSL_psc*Init*() before calling this function.

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

- CSL_SOK Valid PSC handle is returned
- CSL_ESYS_FAIL The PSC instance is invalid

2. Psc object structure is populated.

Modifies
Psc object structure and pStatus variable

Example

```
CSL_Status              status;
CSL_PscObj              pscObj;
CSL_PscHandle           hPsc;
...
hPsc = CSL_pscOpen(&pscObj, CSL_PSC, NULL, &status);
...
```

17.2.3 CSL_pscClose

CSL_Status CSL_pscClose (CSL_PscHandle *hPsc*)

Description
This function closes the specified instance of PSC. CSL for the PSC instance need to be reopened before using any PSC CSL API.

Arguments

hPsc	Pointer to the object that holds reference to the instance of PSC requested after the call

Return Value
CSL_Status

- CSL_SOK - PSC close successful
- CSL_ESYS_BADHANDLE - The handle passed is invalid

Pre Condition
Both *CSL_pscInit()* and *CSL_pscOpen()* must be called successfully in order before calling *CSL_pscClose()*.

Post Condition
The PSC CSL APIs can not be called until the PSC CSL is reopened again using CSL_pscOpen()

Modifies
Obj structure values for the instance

Example

```
CSL_PscHandle           hPsc;
...
CSL_pscClose(hPsc);
```

17.2.4 CSL_pscHwControl

CSL_Status CSL_pscHwControl (CSL_PscHandle ***hPsc***,
CSL_PscHwControlCmd ***cmd***,
void * ***arg***
)

Description
This function performs various control operations on the PSC instance, based on the command passed.

Arguments

hPsc	Handle to the PSC instance
cmd	Operation to be performed on the PSC
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 parameters.

Pre Condition
Both *CSL_pscInit()* and *CSL_pscOpen()* must be called successfully in order before calling this function.

Post Condition
Registers of the PSC instance are configured according to the command and the command arguments. The command determines which registers are modified.

Modifies
PSC Registers

Example

```
CSL_Status     status;
CSL_PscHandle  hPsc;
CSL_PscObj     pscObj;

CSL_pscInit(NULL);
hPsc = CSL_pscOpen(&pscObj, CSL_PSC, NULL, &status);

...
status = CSL_pscHwControl(hPsc,CSL_PSC_CMD_ENABLE_MODULE,NULL);
...
```

17.2.5 CSL_pscGetHwStatus

CSL_Status CSL_pscGetHwStatus (CSL_PscHandle *hPsc*,
CSL_PscHwStatusQuery *query*,
void * *response*
)

Description
This function is used to get the value of various parameters of the PSC instance. The value returned depends on the query passed.

Arguments

hPsc	Handle to the PSC 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
Both *CSL_pscInit()* and *CSL_pscOpen()* must be called successfully in order before calling this function.

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_Status      status;
CSL_PscHandle   hPsc;
CSL_PscObj      pscObj;
Uint32          count;

CSL_pscInit(NULL);
hPsc = CSL_pscOpen(&pscObj, CSL_PSC, NULL, &status);

...

status = CSL_pscGetHwStatus(hPsc, CSL_PSC_QUERY_MODULE_STAT,
                            &count);
...
```

17.2.6 CSL_pscGetBaseAddress

CSL_Status CSL_pscGetBaseAddress (CSL_InstNum *pscNum*,
CSL_PscParam * *pPscParam*,
CSL_PscBaseAddress * *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_pscOpen() 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

pscNum	Specifies the instance of the PSC to be opened
pPscParam	Psc 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 PSC
- CSL_ESYS_FAIL - PSC 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_PscBaseAddress    baseAddress;

...
status = CSL_pscGetBaseAddress(CSL_PSC, NULL, &baseAddress);
...
```

17.3 Data Structures

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

17.3.1 CSL_PscObj

Detailed Description
PSC object structure.

Field Documentation

CSL_InstNum CSL_PscObj::perNum
Instance of PSC being referred by this object

CSL_PscRegsOvly CSL_PscObj::regs
Pointer to the register overlay structure of the PSC

17.3.2 CSL_PscContext

Detailed Description
Module specific context information. Present implementation of PSC CSL doesn't have any context information.

Field Documentation

Uint16 CSL_PscContext::contextInfo
Context information of PSC CSL. The declaration is just a placeholder for future implementation.

17.3.3 CSL_PscParam

Detailed Description
Module specific parameters. Present implementation of PSC CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_PscParam::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

17.3.4 CSL_PscBaseAddress

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

Field Documentation

CSL_PscRegsOvly CSL_PscBaseAddress::regs
Base-address of the configuration registers of the peripheral

17.3.5 CSL_PscmoduleState

Detailed Description
This provides the status of the specified module.

Field Documentation

CSL_PscPeripherals CSL_PscmoduleState:: module
Module to be selected

CSL_PscPeriState CSL_PscmoduleState:: state
Status of the module

17.3.6 CSL_ PscPwrdmnState

Detailed Description
This provides the status of the specified Power Domain.

Field Documentation

CSL_PscPowerDomain CSL_PscPwrdmnState:: pwrDmn
Power Domain to be selected

Uint32CSL_PscPwrdmnState:: State
Status of the Power Domain

17.4 Enumerations

This section lists the enumerations available in the PSC module.

17.4.1 CSL_PscHwControlCmd

enum CSL_PscHwControlCmd
This enum describes the commands used to control the PSC through CSL_pscHwControl().

Enumeration values:

CSL_PSC_CMD_ENABLE_MODULE Enable clock for the specified module.
Parameters:
*CSL_PscPeripherals**

CSL_PSC_CMD_DISABLE_MODULE Disable clock for the specified module
Parameters:
*CSL_PscPeripherals**

CSL_PSC_CMD_PWRDMN_TRNS Enable Power domain-n GO transition
Parameters:
CSL_PscPowerDomain

17.4.2 CSL_PscHwStatusQuery

enum CSL_PscHwStatusQuery
This enum describes the commands used to get status of various parameters of the PSC. These values are used in CSL_pscGetHwStatus().

Enumeration values:

CSL_PSC_QUERY_MODULE_STAT Gets the current status of the specified module.
Parameters:
*CSL_PscmoduleState**

CSL_PSC_QUERY_PWRDMN_TRANS_STAT Gets the transition status of the selected power domain.
Parameters:
*CSL_PscPowerDomain **

17.4.3 CSL_PscPeripherals

enum CSL_PscPeripherals
This provides the name of peripherals controlled by the PSC module.

Enumeration values:

CSL_PSC_MODULE_GEM0	GEM_0 module
CSL_PSC_MODULE_GEM1	GEM_1 module
CSL_PSC_MODULE_GEM2	GEM_2 module
CSL_PSC_MODULE_GEM3	GEM_3 module
CSL_PSC_MODULE_GEM4	GEM_4 module
CSL_PSC_MODULE_GEM5	GEM_5 module
CSL_PSC_MODULE_SRIO	SRIO module
CSL_PSC_MODULE_EMAC0	EMAC0 module
CSL_PSC_MODULE_EMAC1	EMAC1 module
CSL_PSC_MODULE_TSIP0	TSIP0 module
CSL_PSC_MODULE_TSIP1	TSIP1 module
CSL_PSC_MODULE_TSIP2	TSIP2 module
CSL_PSC_MODULE_ HPI	HPI module
CSL_PSC_MODULE_ UTOPIA	UTOPIA module

17.4.4 CSL_PscPowerDomain

enum CSL_PscPowerDomain
This provides the Power Domains on the SOC.

Enumeration values:

CSL_PSC_PWRDMN_ALWAYSON	Always on power domain
CSL_PSC_PWRDMN_PROXYA	Proxy A power domain
CSL_PSC_PWRDMN_PROXYB	Proxy B power domain
CSL_PSC_PWRDMN_PROXYC	Proxy C power domain
CSL_PSC_PWRDMN_PROXYD	Proxy D power domain
CSL_PSC_PWRDMN_PROXYE	Proxy E power domain
CSL_PSC_PWRDMN_PROXYF	Proxy F power domain
CSL_PSC_PWRDMN_PROXYG	Proxy G power domain
CSL_PSC_PWRDMN_PROXYH	Proxy H power domain

17.4.5 CSL_PscPeriState

enum CSL_PscPeriState
This enum describes the state of the module.

Enumeration values:

CSL_PSC_MODULE_SWRST_DISABLE	module software reset disable
CSL_PSC_MODULE_SYNCRST	Module sync reset
CSL_PSC_MODULE_DISABLE	Module disable
CSL_PSC_MODULE_ENABLE	Module enable

17.5 Typedefs

typedef CSL_PscObj * CSL_PscHandle
This is a pointer to CSL_PscObj and is passed as the first parameter to all PSC CSL APIs

Chapter 18
SMC Module

Topics

18.1 Overview

Shared memory controller is interfaced to GEM via its UMC's UMAP1 memory port. The control interface between UMC and SMC is Pipeline-Ack based. All the traffic either from L1P, or L1D, or EMC to SMC comes via UMC.

SMC operates at half the CPU's frequency (CLK2).
SMC logic is divided in to two logics, one of them is "per-GEM SMC logic" that physically sits near the GEM boundary and second is "per-BANK SMC logic" that is shared by all the GEMs connected to SMC.

The SMC connects to M 256-bit wide memory banks and implements LS-banking for its memories.

To implement LS-banking, number of memory banks (M) connected to SMC must be power of 2 (i.e. M = 2, 4, 8 and so on).

18.2 Functions

This section lists functions available in the SMC module.

18.2.1 CSL_smcInit

CSL_Status CSL_smcInit (CSL_SmcContext * pContext)

Description
This function is idempotent i.e. calling it many times is same as calling it once. This function is only for book-keeping purpose and it doesn't touch the hardware (read/write registers) in any manner.

Arguments

```
pContext    Context information for the instance. Should be NULL
```

Return Value
`CSL_Status`

- `CSL_SOK` - Always returns

Pre Condition
None

Post Condition
The CSL for SMC is initialized

Modifies
None

Example

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

18.2.2 CSL_smcOpen

```
CSL_SmcHandle CSL_smcOpen (   CSL_SmcObj *        pSmcObj,
                              CSL_InstNum          smcNum,
                              CSL_SmcParam *       pSmcParam,
                              CSL_Status *         pStatus
                              )
```

Description
Reserves the specified SMC for use. The device can be re-opened anytime after it has been normally closed, if so required. The SMC handle returned by this call is input as an essential argument for the rest of the APIs in SMC module.

Arguments

```
pSmcObj     Pointer to SMC object that holds the context.
```

	Memory for this object should be allocated by the user
smcNum	The instance of SMC to be opened
pSmcParam	Parameter for SMC
pStatus	Pointer to the variable that holds the status of the open call

Return Value
CSL_SmcHandle

- CSL_SmcHandle to the requested instance of SMC if the call is successful, otherwise NULL is returned.

Pre Condition
CSL_smcInit() must be called successfully. Memory for the CSL_SmcObj must be allocated outside this call. This object must be retained while using this peripheral instance.

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

CSL_SOK - Valid SMC handle is returned
CSL_ESYS_FAIL - The SMC instance is invalid
CSL_ESYS_INVPARAMS - Invalid parameter

2. SMC object structure is populated.

Modifies
1. The status variable
2. object structure

Example

```
CSL_SmcHandle     hSmc;
CSL_SmcObj        smcObj;
CSL_Status        status;
...
hSmc = CSL_smcOpen(&smcObj, CSL_SMC_0, NULL, &status);
...
```

18.2.3 CSL_smcClose

CSL_Status CSL_smcClose (CSL_SmcHandle hSmc)

Description
Unreserves the SMC identified by the handle passed.

Arguments

hSmc	Handle to the SMC

Return Value
CSL_Status

- CSL_SOK - Close Successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both CSL_smcInit() and CSL_smcOpen() must be called successfully in order before calling CSL_smcClose().

Post Condition
None

Modifies
SMC Handle

Example

```
CSL_Status        status;
CSL_smcHandle     hSmc;
...
status = CSL_smcClose(hSmc);
...
```

18.2.4 CSL_smcGetBaseAddress

CSL_Status CSL_smcGetBaseAddress (CSL_InstNum smcNum, CSL_SmcParam * pSmcParam, CSL_SmcBaseAddress * 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_smcOpen() 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. MMR's go to an alternate location.

Arguments

smcNum	Specifies the instance of the SMC to be opened.
pSmcParam	Module specific parameters.
pBaseAddress	Pointer to baseaddress structure containing base address details

Return Value
CSL_Status

- CSL_SOK - Open call is successful
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Inavlid 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_SmcBaseAddress      baseAddress;
...
status = CSL_smcGetBaseAddress(CSL_SMC_0,
                                        NULL,
                                    &baseAddress);
```

18.2.5 CSL_smcGetHwStatus

CSL_Status CSL_smcGetHwStatus (	**CSL_SmcHandle**	**hSmc,**
	CSL_SmcHwStatusQuery	**query,**
	void *	**response**
	)	

Description
Gets the status of different operations or some setup-parameters of SMC. The status is returned through the third parameter.

Arguments

hSmc	SMC handle returned by successful 'open'
query	The query to this API of SMC which indicates the status to be returned. Query command, refer @a CSL_SmcHwStatusQuery for the list of query commands supported
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 - Status info return successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Invalid query command
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both CSL_smcInit() and CSL_smcOpen() must be called successfully in order before calling CSL_smcGetHwStatus(). Refer to CSL_SmcHwStatusQuery for the argument to be passed along with the corresponding query command.

Post Condition
None

Modifies
Third parameter

Example

```
CSL_SmcHandle hSmc;
CSL_Status status;
Uint32 response;
...
status = CSL_smcGetHwStatus(hSmc,CSL_SMC_QUERY_MODE,&response);
...
```

18.2.6 CSL_smcHwControl

CSL_Status CSL_smcHwControl (CSL_SmcHandle **hSmc,**
CSL_SmcControlCmd **cmd,**
void * arg
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of SMC.

Arguments

hSmc	SMC handle returned by successful 'open'
cmd	The command to this API indicates the action to be taken on SMC. Control command, refer @a CSL_SmcControlCmd for the list of commands supported
arg	An optional argument. Optional argument as per the control command, @a void * casted

Return Value
CSL_Status

- CSL_SOK - Status info return successful.
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVCMD - Invalid command

Pre Condition
Both CSL_smcInit() and CSL_smcOpen() must be called successfully in order before calling CSL_smcHwControl().

Refer to CSL_SmcHwControlCmd for the argument type (void*) that needs to be passed with the control command

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

Modifies
The hardware registers of SMC.

Example

```
CSL_Status        status;
Uint32            arg;
CSL_SmcHandle     hSmc;
...
// SMC object defined and HwSetup structure defined and
initialized
...
// Init successfully done
...
// Open successfully done
...
arg = CSL_SMC_PREFETECH_PAGE(1);
status = CSL_smcHwControl(    hSmc,
                              CSL_SMC_CMD_ENA_PREFETCH_PAGE,
                              &arg);
```

18.3 Data Structures

This section lists Data Structures available in the SMC module.

18.3.1 CSL_SmcBaseAddress

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

Field Documentation

CSL_SmcRegsOvly CSL_SmcBaseAddress::regs
Base-address of the Configuration registers of SMC.

18.3.2 CSL_SmcContext

Detailed Description
SMC specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_SmcContext::contextInfo
Context information of SMC.

18.3.3 CSL_SmcData

Detailed Description
This structure is used to hold the configuration/status information of different SMC pages.

Field Documentation

CSL_BitMask32 CSL_SmcData::data
Desired information in the registers

Uint32 CSL_SmcData::index
Page selection

18.3.4 CSL_SmcObj

Detailed Description
This structure/object holds the context of the instance of SMC opened using CSL_smcOpen() function.
Pointer to this object is passed as SMC Handle to all SMC CSL APIs. CSL_smcOpen() function initializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_SmcObj::perNum
Instance of SMC being referred by this object

CSL_SmcRegsOvly CSL_SmcObj::regs
Pointer to the register overlay structure of the SMC

18.3.5 CSL_SmcParam

Detailed Description
SMC specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_SmcParam::flags
Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

18.4 Enumerations

18.4.1 CSL_SmcControlCmd

enum CSL_SmcControlCmd
This is the set of control commands that are passed to CSL_smcHwControl(), 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_SMC_CMD_ENA_PREFETCH_PAGE	Setup to enable the prefetch page **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_DIS_PREFETCH_PAGE	Setup to disable the prefetch page **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_SET_PREFETCH_FLUSH	Setup the prefetch flush **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_CLR_MPFSR_MPFAR	Setup to clear the MPFSR and MPFAR **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_SET_SLEEP_PAGE2	Setup to page2 sleep **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_SET_WAKE_PAGE2	Setup to page2 wake **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_SET_SLEEP_PAGE3	Setup to page3 sleep **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMC_CMD_SET_WAKE_PAGE3	Setup to page3 wake **Parameters**:*None* **Returns**:CSL_SOK

18.4.2 CSL_SmcHwStatusQuery

enum CSL_SmcHwStatusQuery
This is the set of query commands to get the status of various operations in SMC
The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_SMC_QUERY_PREFETCH_PAGE_STS	Queries the prefetech page status **Parameters**:*(Uint32 *)*

	Returns:CSL_SOK
CSL_SMC_QUERY_MODE	Queries the mode **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_CPU_ID	Queries the Cpu Id **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_FAULT_ADDRESS	Queries the Fault Address **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_PWRDWN_PAGE_STS	Queries the powerdown page status **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_AML_LINK_DATA_STS	Queries the AML link data status **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_ATOMIC_LINK	Queries the atomic link **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_LINK_OWNER	Queries the link owner status **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_LINK_ADDR	Queries the link address **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMC_QUERY_AML_LINK_DATA	Queries the link data **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK

18.5 Macros

#define CSL_SMC_POWERDOWN_PAGE2 (0x00000004u)
to set/reset page2

#define CSL_SMC_POWERDOWN_PAGE3 (0x00000008u)
to set/reset page3

#define CSL_SMC_PREFETECH_PAGE (x) (1 << x)
To enable/disable specific pages

#define CSL_SMC_PREFETECH_PAGES_ALL (0xFFFFFFFFu)
To enable/disable all the pages

18.6 Typedefs

typedef CSL_SmcObj CSL_SmcObj
This structure/object holds the context of the instance of SMC opened using CSL_smcOpen() function.

typedef CSL_SmcObj * CSL_SmcHandle
This is a pointer to CSL_SmcObj and is passed as the first parameter to all SMC CSL APIs.

Chapter 19
SMCP Module

Topics

19.1 Overview

Captures GEM read accesses to SMC and calculates their performance.

- Capture the Prefetch triggered events from SMC for the corresponding GEM to calculate the energy efficiency of the SMC.
- Separate Config bus SCR interface to GEM/s.

In Multi GEM devices that uses shared memory, like Tomahawk, the performance statistics of a single GEM in the shared memory is required to optimize the device performance in the system. The logic to calculate the performance of a GEM in shared memory can be implemented in the shared memory controller (SMC) but on the cost of SMC design and verification complexity.

19.2 Functions

This section lists functions available in the SMCP module.

19.2.1 CSL_smcpInit

CSL_Status CSL_smcpInit (CSL_SmcpContext * pContext)

Description
This function is idempotent i.e. calling it many times is same as calling it once. This function is only for book-keeping purpose and it doesn't touch the hardware (read/write registers) in any manner.

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_smcpInit(NULL);
```

19.2.2 CSL_smcpOpen

```
CSL_SmcpHandle CSL_smcpOpen (   CSL_SmcpObj *      pSmcpObj,
                                CSL_InstNum        smcpNum,
                                CSL_SmcpParam *    pSmcpParam,
                                CSL_Status *       pStatus
                                )
```

Description
Reserves the specified SMCP for use. The device can be re-opened anytime after it has been normally closed, if so required. The SMCP handle returned by this call is input as an essential argument for the rest of the APIs in SMCP module.

Arguments

```
pSmcpObj          Pointer to SMCP object that holds the context.
                  Memory for this object should be allocated by
                  the user
```

smcpNum	The instance of SMCP to be opened
pSmcpParam	Parameter for SMCP
pStatus	Pointer to the variable that holds the status of the open call

Return Value
CSL_SmcpHandle

- CSL_SmcpHandle to the requested instance of SMCP if the call is successful, otherwise NULL is returned.

Pre Condition
CSL_smcpInit() must be called successfully. Memory for the CSL_SmcpObj must be allocated outside this call. This object must be retained while using this peripheral instance.

Post Condition

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

CSL_SOK - Valid SMCP handle is returned
CSL_ESYS_FAIL - The SMCP instance is invalid
CSL_ESYS_INVPARAMS - Invalid parameter

2. SMCP object structure is populated.

Modifies
1. The status variable
2. object structure

Example

```
CSL_SmcpHandle    hSmcp;
CSL_SmcpObj       smcpObj;
CSL_Status        status;
...
hSmcp = CSL_smcpOpen(&smcpObj, CSL_SMCP_0, NULL, &status);
...
```

19.2.3 CSL_smcpClose

CSL_Status CSL_smcpClose (CSL_SmcpHandle hSmcp)

Description
Unreserves the SMCP identified by the handle passed.

Arguments

hSmcp	Handle to the SMCP

Return Value
CSL_Status

- CSL_SOK - Close Successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both CSL_smcpInit() and CSL_smcpOpen() must be called successfully in order before calling CSL_smcpClose().

Post Condition
None

Modifies
SMCP Handle

Example

```
CSL_Status        status;
CSL_smcpHandle    hSmcp;
...
status = CSL_smcpClose(hSmcp);
...
```

19.2.4 CSL_smcpGetBaseAddress

CSL_Status CSL_smcpGetBaseAddress (CSL_InstNum smcpNum, CSL_SmcpParam * pSmcpParam, CSL_SmcpBaseAddress * 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_smcpOpen() 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. MMR's go to an alternate location.

Arguments

smcpNum	Specifies the instance of the SMCP to be opened.
pSmcpParam	Module specific parameters.
pBaseAddress	Pointer to baseaddress structure containing base address details.

Return Value
CSL_Status

- CSL_SOK - Open call is successful
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Inavlid 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_SmcpBaseAddress     baseAddress;
...
status = CSL_smcpGetBaseAddress(CSL_SMCP_0,
                                NULL,
                                &baseAddress);
...
```

19.2.5 CSL_smcpGetHwStatus

CSL_Status CSL_smcpGetHwStatus (	**CSL_SmcpHandle**	**hSmcp,**
	CSL_SmcpHwStatusQuery	**query,**
	void *	**response**
	)	

Description
Gets the status of different operations or some setup-parameters of SMCP. The status is returned through the third parameter.

Arguments

hSmcp	SMCP handle returned by successful 'open'
query	The query to this API of SMCP which indicates the status to be returned. Query command, refer @a CSL_SmcpHwStatusQuery for the list of query commands supported
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 - Status info return successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Invalid query command
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both CSL_smcpInit() and CSL_smcpOpen() must be called successfully in order before calling CSL_smcpGetHwStatus(). Refer to CSL_SmcpHwStatusQuery for the argument to be passed along with the corresponding query command.

Post Condition
None

Modifies
Third parameter.

Example

```
CSL_SmcpHandle    hSmcp;
CSL_Status        status;
Uint32            response;
...
status = CSL_smcpGetHwStatus( hSmcp,
                              CSL_SMCP_QUERY_PFC_CNT,
                              &response);
...
```

19.2.6 CSL_smcpHwControl

CSL_Status CSL_smcpHwControl (	**CSL_SmcpHandle**	**hSmcp,**
	CSL_SmcpControlCmd	**cmd,**
	void *	**arg**
	)	

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of SMCP.

Arguments

hSmcp	SMCP handle returned by successful 'open'
cmd	The command to this API indicates the action to be taken on SMCP. Control command, refer @a CSL_SmcpControlCmd for the list of commands supported
arg	An optional argument. Optional argument as per the control command, @a void * casted

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
Both CSL_smcpInit() and CSL_smcpOpen() must be called successfully in order before calling CSL_smcpHwControl().

Refer to CSL_SmcpHwControlCmd for the argument type (void*) that needs to be passed with the control command

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

Modifies
The hardware registers of SMCP.

Example

```
CSL_Status        status;
Uint32            arg;
CSL_SmcpHandle    hSmcp;
...
// SMCP object defined and HwSetup structure defined and
initialized
...
// Init successfully done
...
// Open successfully done
...
arg = CSL_SMCP_BANK0;
status = CSL_smcpHwControl(hSmcp,CSL_SMCP_CMD_SET_BANK,&arg);
```

19.3 Data Structures

This section lists Data Structures available in the SMCP module.

19.3.1 CSL_SmcpBaseAddress

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

Field Documentation

CSL_SmcpRegsOvly CSL_SmcpBaseAddress::regs
Base-address of the Configuration registers of SMCP.

19.3.2 CSL_SmcpContext

Detailed Description
SMCP specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_SmcpContext::contextInfo
Context information of SMCP. The below declaration is just a place-holder for future implementation.

19.3.3 CSL_SmcpData

Detailed Description
This structure is used to hold the configuration/status information of different SMC pages.

Field Documentation

CSL_BitMask32 CSL_SmcpData::data
Desired information in the registers

Uint32 CSL_SmcpData::index
Page selection

19.3.4 CSL_SmcpObj

Detailed Description
This structure/object holds the context of the instance of SMCP opened using CSL_smcpOpen() function.
Pointer to this object is passed as SMCP Handle to all SMCP CSL APIs. CSL_smcpOpen() function initializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_SmcpObj::perNum
Instance of SMCP being referred by this object

CSL_SmcpRegsOvly CSL_SmcpObj::regs
Pointer to the register overlay structure of the SMCP

19.3.5 CSL_SmcpParam

Detailed Description
SMCP specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_SmcpParam::flags
Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

19.4 Enumerations

19.4.1 CSL_SmcpControlCmd

enum CSL_SmcpControlCmd
This is the set of control commands that are passed to CSL_smcpHwControl(), 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_SMCP_CMD_SET_BANK	Setup to set the bank **Parameters**: *None* **Returns**:CSL_SOK
CSL_SMCP_CMD_CLR_PWS_CNT	Setup the CLEAR bit of SL2PCMD **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMCP_CMD_ENA_DIS_PROFILER	Setup to enable / disable the profiler **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMCP_CMD_SET_MASK_WS	Setup combine WSx_EVENT to generate SMC profile event to GEM **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMCP_CMD_RESET_MASK_WS	Setup does not combine WSx_EVENT to generate SMC profile event to GEM **Parameters**:*None* **Returns**:CSL_SOK
CSL_SMCP_CMD_RESET_BANK	Setup to reset the bank **Parameters**:*None* **Returns**:CSL_SOK

19.4.2 CSL_SmcpHwStatusQuery

enum CSL_SmcpHwStatusQuery
This is the set of query commands to get the status of various operations in SMCP
The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_SMCP_QUERY_BANK	Queries the bank **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_PWS_CNT	Queries the CLEAR bit of SL2PCMD **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK

CSL_SMCP_QUERY_PFC_CNT	Queries the wait x read state accesses **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_PWS_CNT_SAT	Queries the counter saturation state **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_PFC_CNT_SAT	Queries the prefetech count saturate state **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_PROFILER_STS	Queries the profiler status **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_MASK_WS	Queries the wait read access state **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_SMCP_QUERY_PWS_CNT_CLR	Queries the prefetch wait read access clear status **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK

19.5 Macros

#define CSL_SMCP_BANK0 (0x00000001u)
Mask value for bank0

#define CSL_SMCP_BANK1 (0x00000002u)
Mask value for bank1

#define CSL_SMCP_BANK2 (0x00000004u)
Mask value for bank2

#define CSL_SMCP_BANK3 (0x00000008u)
Mask value for bank3

#define CSL_SMCP_MASK_WS_0 (0x00000001u)
SMC read request was 0 wait state read request.

#define CSL_SMCP_MASK_WS_1 (0x00000002u)
SMC read request was 1 wait state read request.

#define CSL_SMCP_MASK_WS_2 (0x00000004u)
SMC read request was 2 wait state read request.

#define CSL_SMCP_MASK_WS_3 (0x00000008u)
SMC read request was 3 wait state read request.

#define CSL_SMCP_MASK_WS_4 (0x00000010u)
SMC read request was 4 wait state read request.

#define CSL_SMCP_MASK_WS_5 (0x00000020u)
SMC read request was 5 wait state read request.

#define CSL_SMCP_MASK_WS_6 (0x00000040u)
SMC read request was 6 wait state read request.

#define CSL_SMCP_MASK_WS_7 (0x00000080u)
SMC read request was 7 wait state read request.

19.6 Typedefs

typedef CSL_SmcpObj CSL_SmcpObj
This structure/object holds the context of the instance of SMCP opened using CSL_smcpOpen() function.

typedef CSL_SmcpObj * CSL_SmcpHandle
This is a pointer to CSL_SmcpObj and is passed as the first parameter to all SMCP CSL APIs.

Chapter 20
SRIO Module

Topics

20.1 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
- 2X Serial RapidIO with auto-negotiation to 1X port, optional operation for (2) 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.

20.2 Functions

This section lists the functions available in the SRIO module.

20.2.1 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_srioInit(NULL);
```

20.2.2 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.

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 parameters.

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);
...
```

20.2.3 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);
```

20.2.4 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_SRIO_HWSETUP_DEFAULTS;
CSL_Status              status;
CSL_SrioControlSetup       periSetup;
CSL_SrioBlkEn                 blockSetup;
CSL_SrioPktFwdCntl            pktFwdSetup;


hSrio = CSL_srioOpen (&srioObj, CSL_SRIO, NULL, &status);

status = CSL_srioHwSetup(hSrio, &hwSetup);
```

20.2.5 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

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;
CSL_Status       status;
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);
...
```

20.2.6 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 parameters

Pre Condition

Both *CSL_srioInit()* and *CSL_srioOpen()* must be called successfully in order before calling this API.

Post Condition

Data requested by the query is returned through the variable "response".

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);
...
```

20.2.7 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);
...
```

20.2.8 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 parameters

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_Status          status;
CSL_SrioHwSetup     hwSetup;
...
status = CSL_srioGetHwsetup(hSrio, &hwSetup);
...
```

20.2.9 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
None

Post Condition
The LSU module registers are configured with the passed parameters and the data transfer starts.

Modifies
LSU module registers

Example

```
#define LARGE_DEV_ID          0xBEEF
#define SRIO_PKT_TYPE_NWRITE  0x54
CSL_Status                    status;
CSL_SrioDirectIO_ConfigXfr    lsuConfig;
Uint8                         index;
Uint32                        dst=0x20000000;
Uint32                        src=0x30000000;
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                 = LARGE_DEV_ID;
    lsuConfig.intrReq               = 0;   /* No interrupts */
    lsuConfig.pktType               = SRIO_PKT_TYPE_NWRITE;
                                      /* 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);
```

20.2.10 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 - Open call is successful
- CSL_ESYS_FAIL - SRIO 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_SrioBaseAddress baseAddress;
...
status = CSL_SrioGetBaseAddress(CSL_SRIO, NULL, &baseAddress);
...
```

20.3 Data Structures

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

20.3.1 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

20.3.2 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[CSL_SRIO_PORTS_MAX]
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[CSL_SRIO_PORTS_MAX]
Packet forwarding registers for 16-bit and 8-bit device IDs

Uint32 CSL_SrioConfig::INTDST_RATE_CNTL[6]
INTDST interrupt rate control register for DST 0

Uint32 CSL_SrioConfig::IP_PRESCALAR
Serial port IP prescalar

CSL_SrioCfgLsuRegs CSL_SrioConfig::LSU[CSL_SRIO_PORTS_MAX]
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[CSL_SRIO_PORTS_MAX]
Port registers

CSL_SrioCfgPortErrorRegs CSL_SrioConfig::PORT_ERROR[CSL_SRIO_PORTS_MAX]
Port error CSR

CSL_SrioCfgPortOptionRegs CSL_SrioConfig::PORT_OPTION[CSL_SRIO_PORTS_MAX]
Port options CSR

Uint32 CSL_SrioConfig::PW_TGT_ID
Port-write target device ID CSR

Uint32 CSL_SrioConfig::SERDES_CFG_CNTL
SerDes macros configuration control registers

Uint32 CSL_SrioConfig::SERDES_CFGRX_CNTL[CSL_SRIO_PORTS_MAX]
SerDes RX channels configuration control registers

Uint32 CSL_SrioConfig::SERDES_CFGTX_CNTL[CSL_SRIO_PORTS_MAX]
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

20.3.3 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.

20.3.4 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

Uint32 CSL_SrioHwSetup::deviceId3
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

Uint32 CSL_SrioHwSetup::deviceId4
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[CSL_SRIO_PORTS_MAX]
Sets the boundaries for device IDs that are part of the chain and the packet can be forwarded to

CSL_SrioPortCntlIndpEn CSL_SrioHwSetup::portCntlIndpEn[CSL_SRIO_PORTS_MAX]
Port control independent error reporting enable. Macros can be OR'ed to get the value

CSL_SrioPortCntlConfig CSL_SrioHwSetup::portCntlSetup[CSL_SRIO_PORTS_MAX]
Port control configuration

CSL_SrioPortErrConfig CSL_SrioHwSetup::portErrSetup[CSL_SRIO_PORTS_MAX]
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 stopped by software writing to the error detect registers

Bool CSL_SrioHwSetup:: serDesLoopback
SERDES loopback

Uint32 CSL_SrioHwSetup:: serDesRxChannelCfg [CSL_SRIO_PORTS_MAX]
SERDES RX channel configure

Uint32 CSL_SrioHwSetup::serDesPllCfg
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_SrioSilenceTimer **CSL_SrioHwSetup::silenceTimer[CSL_SRIO_PORTS_MAX]**
Silence timer. Defines the time of the port in the SILENT state

Uint32 CSL_SrioHwSetup:: serDesTxChannelCfg [CSL_SRIO_PORTS_MAX]
SERDES TX channel configure

20.3.5 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.

20.3.6 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

20.3.7 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

20.3.8 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

20.3.9 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

20.3.10 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

20.3.11 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::logicalLayerDisable
Logical layer disable. This bit disables all the packet types at the logical layer. 0 - All non-matching packets are destroyed. 1 - All packets, regardless of the destination ID, are forwarded to the application.

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

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

20.3.12 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

20.3.13 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

20.3.14 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

20.3.15 CSL_SrioSpErrDetStat

Detailed Description
This structure is used to clear port error detect status.

Field Documentation

Bool CSL_SrioSpErrDetStat::illTransErr
Illegal transfer error. 0 - not detected, 1 - detected. Write 1 to clear.

Bool CSL_SrioSpErrDetStat::irqErr
Interrupt error status.
0 - an error has not occurred and/or there is not a Port-Write condition.
1 - An error occurred and there is a Port-Write condition. Write 1 to clear.

Bool CSL_SrioSpErrDetStat::maxRetryErr
Maximum retry error. 0 - no error condition detected, - max_retry_cnt is equal to max_retry_threshold. Write 1 to clear.

CSL_SrioPortNum CSL_SrioSpErrDetStat::portNum
Port number

20.3.16 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

20.3.17 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

20.3.18 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

20.3.19 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.

20.3.20 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

20.3.21 CSL_SrioPortCntlIndpEn

Detailed Description
This structure contains configuration in port control independent register.

Field Documentation

Bool CSL_SrioPortCntlIndpEn::debug
Mode of operation. 0 - normal mode, 1 - debug mode.

Bool CSL_SrioPortCntlIndpEn:: forceReInit
Force reinitialization process. 0 - do not force reinitialization, 1 - force reinitialization.

Bool CSL_SrioPortCntlIndpEn::illTransEn
Illegal transfer error reporting. 0 - disable, 1 - enable.

Bool CSL_SrioPortCntlIndpEn::irqEn
Interrupt error reporting. 0 - disable, 1 - enable.

Bool CSL_SrioPortCntlIndpEn::maxRetryEn
Maximum retry error reporting. 0 - disable, 1 - enable.

Uint8 CSL_SrioPortCntlIndpEn::maxRetryThr
The threshold value for reporting an error condition. 0 - disable the max_retry_error reporting, n (>0) - set the max_retry_threshold to n.

Bool CSL_SrioPortCntlIndpEn::sendDebugPkt
Send debug packet. Write 1 to force the sending of a debug packet. For debug mode only.

Bool CSL_SrioPortCntlIndpEn::softRecovery
Software controlled error recovery.
0 - Transmission of error recovery sequence is performed by the hardware,
1 - Transmission of error recovery sequence is performed by the software.

20.3.22 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::srioCustom
Identifies the custom field of SRIO

Uint8 CSL_SrioPidNumber::srioFunc
Identifies the FUNC field of SRIO

Uint8 CSL_SrioPidNumber::srioMajor
Identifies the major version of SRIO

Uint8 CSL_SrioPidNumber::srioMinor
Identifies the minor version of SRIO

Uint8 CSL_SrioPidNumber::srioRtl
Identifies the RTL of SRIO

Uint8 CSL_SrioPidNumber::srioScheme
Identifies the scheme of SRIO

20.3.23 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)

20.3.24 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

20.3.25 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

20.3.26 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

Uint32 CSL_SrioLsuCompStat::lsuNum
LSU number

20.3.27 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

20.3.28 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

20.3.29 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

20.3.30 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

20.3.31 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

20.3.32 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

Bool CSL_SrioSerDesRxCfg::enTest
Enable test mode

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.

20.3.33 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::enTest

Enable test mode

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

20.4 Enumerations

This section lists the enumerations available in the SRIO module.

20.4.1 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_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:** *CSL_SrioPortData**

20.4.2 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*

20.4.3 CSL_SrioPortCaptType

enum CSL_SrioPortCaptType
This enum describes type of the captured information type 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 during error

20.4.4 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

20.4.5 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

20.4.6 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

20.4.7 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

20.4.8 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

20.4.9 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

20.4.10 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

20.4.11 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)

20.4.12 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

20.4.13 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

20.4.14 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

20.4.15 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

20.4.16 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

20.4.17 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

20.4.18 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

20.4.19 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

20.4.20 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.

20.4.21 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

20.4.22 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

20.4.23 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

20.4.24 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

20.4.25 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

20.5 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_ERR_OUTPUT_ERROR_STP (0x00010000)

#define CSL_SRIO_ERR_INPUT_ERROR_STP (0x00000100)
Port error stop status 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_IDLE_ERR_DISABLE (0x20000000)

#define CSL_SRIO_TX_FIFO_BYPASS (0x10000000)

#define CSL_SRIO_PW_DISABLE (0x08000000)

#define CSL_SRIO_SRC_TGT_ID_DISABLE (0x04000000)

#define CSL_SRIO_SELF_RESET (0x02000000)

#define CSL_SRIO_F8_TGT_ID_DISABLE (0x01000000)

#define CSL_SRIO_MLTC_ENABLE (0x00000020)

#define CSL_SRIO_RST_ENABLE (0x00000008)

#define CSL_SRIO_PW_ENABLE (0x00000002)
Port ip mode macros

#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

```
/** SRIO Packet type (of 8bits) definition
  * 4 msb = 4b Ftype field for all packets and
  * 4 lsb = 4b trans field (Ttype)for packet types 2,5,8.
  */
/** Ftype=8 and Ttype = 0 */
#define CSL_SRIO_REQ_MAINT_RD 0x80

/** Ftype=8 and Ttype = 1 */
#define CSL_SRIO_REQ_MAINT_WR 0x81

/** Ftype=8 and Ttype = 2 */
#define CSL_SRIO_REQ_MAINT_RD_RESP 0x82

/** Ftype=8 and Ttype = 3 */
#define CSL_SRIO_REQ_MAINT_WR_RESP 0x83

/** Ftype=8 and Ttype = 4 */
#define CSL_SRIO_REQ_MAINT_PW 0x84

/** Ftype=2 and Ttype = 4 */
#define CSL_SRIO_REQ_NREAD 0x24

/** Ftype=2 and Ttype = 12 */
#define CSL_SRIO_REQ_ATOMIC_INC 0x2C

/** Ftype=2 and Ttype = 13 */
#define CSL_SRIO_REQ_ATOMIC_DEC 0x2D

/** Ftype=2 and Ttype = 14 */
#define CSL_SRIO_REQ_ATOMIC_SET 0x2E

/** Ftype=2 and Ttype = 15 */
#define CSL_SRIO_REQ_ATOMIC_CLR 0x2F

/** Ftype=5 and Ttype = 0 */
#define CSL_SRIO_REQ_NWRITE 0x54

/** Ftype=5 and Ttype = 0 */
#define CSL_SRIO_REQ_NWRITE_R 0x55

/** Ftype=5 and Ttype = 0 */
#define CSL_SRIO_REQ_ATOMIC_TNS 0x5E

/** Ftype=6 and Ttype = Don't care */
#define CSL_SRIO_REQ_SWRITE 0x60

/** Ftype=7 and Ttype = Don't care */
```

#define CSL_SRIO_REQ_CONGESTION 0x70

/** Ftype=10 and Ttype = Don't care */
#define CSL_SRIO_REQ_DBLL 0xA0

/** Ftype=11 and Ttype = Don't care */
#define CSL_SRIO_REQ_MSG 0xB0

/** Ftype=13 and Ttype = 0 */
#define CSL_SRIO_REQ_DBLL_RESP 0xD0

/** Ftype=13 and Ttype = 1 */
#define CSL_SRIO_REQ_MSG_RESP 0xD1

/** Ftype=13 and Ttype = 8 */
#define CSL_SRIO_REQ_RESP_WO_PAYLOAD 0xD8

#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 2
Number of ports

#**define CSL_SRIO_HWSETUP_DEFAULTS**

Value:

```
{ \
    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, \
        CSL_SRIO_DEVICEID_REG3_RESETVAL, \
        CSL_SRIO_DEVICEID_REG4_RESETVAL, \
    { \
      {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_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 \
            } \
    }, \
     {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_CONTRO
      L_RESETVAL, \
     {\
        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 \
        } \
    }, \
    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_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_SRIO_SP_CTL_INDEP_RESETVAL, \
         CSL_SRIO_SP_CTL_INDEP_RESETVAL  \
         }\
 }
```

Default hardware setup parameters

#define CSL_SRIO_CONFIG_DEFAULTS

Value:

```
{ \
     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_RESETVAL, \
```

```
CSL_SRIO_DEVICEID_REG3_RESETVAL, \
CSL_SRIO_DEVICEID_REG4_RESETVAL, \      {\
     {0xFFFFFFFF, 0x0003FFFF}, \
     {0xFFFFFFFF, 0x0003FFFF} \
 }, \
 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_INTDST_RATE_CNTL_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_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_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_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 \
        } \
    }\
}
```

Default values for config structure

20.6 Typedefs

typedef CSL_SrioObj* CSL_SrioHandle
This data type is used to return the handle to the CSL of the SRIO.

Chapter 21
TIMER MODULE

Topics

21.1 Overview

This chapter describes the Functions, Data Structures, Enumerations and Macros within TIMER module. This is having twelve 64-bit general-purpose timers. Each of the TIMER peripherals (TIMER0 to TIMER11) is configurable as either 64-bit general-purpose timer or two 32-bit general-purpose timers or a watchdog timer. Each timer is made up of two 32-bit counters: a high counter and a low counter.

The timers can be used to: time events, count events, generate pulses, interrupt the CPU, and send synchronization events to the EDMA.

21.2 Functions

This section lists the functions available in the TIMER module.

21.2.1 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 General purpose 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_tmrInit(NULL);
```

21.2.2 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 twelve instances 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

Timer object structure and pStatus variable

Example

```
CSL_Status              status;
CSL_TmrObj              tmrObj;
CSL_TmrHandle           hTmr;
...
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);
...
```

21.2.3 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 requested after the call

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_tmrClose(hTmr);
```

21.2.4 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;
CSL_TmrObj          tmrObj;

hwSetup.tmrTimerPeriodLo = 0x100;
hwSetup.tmrTimerPeriodLo = 0x100;
...
CSL_tmrInit(NULL);
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);

...
status = CSL_tmrHwSetup(hTmr, &hwSetup);
...
```

21.2.5 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

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
Registers determined by the command

Example

```
CSL_Status     status;
CSL_TmrHandle  hTmr;
CSL_TmrObj     tmrObj;

CSL_tmrInit(NULL);
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);

...
status = CSL_tmrHwControl(hTmr, CSL_TMR_CMD_START_TIMLO, NULL);
...
```

21.2.6 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;
CSL_TmrObj     tmrObj;
Uint32        count;

CSL_tmrInit(NULL);
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);

...

status = CSL_tmrGetHwStatus(hTmr, CSL_TMR_QUERY_COUNT_LO,
                            &count);
...
```

21.2.7 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 General purpose timer instance

Example

```
CSL_TmrHandle          hTmr;
CSL_TmrConfig          config = CSL_TMR_CONFIG_DEFAULTS;
CSL_Status             status;
```

```
CSL_TmrObj              tmrObj;

CSL_tmrInit(NULL);
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);
...
status = CSL_tmrHwSetupRaw(hTmr, &config);
...
```

21.2.8 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 - Invalid Parameters

Pre Condition
Both *CSL_tmrInit()* and *CSL_tmrOpen()* must be called successfully in order before calling this function.

Post Condition
The hardware set up structure will be populated with values from the registers

Modifies
Second parameter *hwSetup* value

Example

```
CSL_Status        status;
CSL_TmrHandle     hTmr;
CSL_TmrHwSetup    hwSetup;
CSL_TmrObj        tmrObj;

CSL_tmrInit(NULL);
hTmr = CSL_tmrOpen(&tmrObj, CSL_TMR_1, NULL, &status);

...
status = CSL_tmrGetHwSetup(hTmr, &hwSetup);
...
```

21.2.9 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);
...
```

21.3 Data Structures

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

21.3.1 CSL_TmrObj

Detailed Description
Watchdog timer object structure.

Field Documentation

CSL_InstNum CSL_TmrObj:: tmrNum
Instance of timer being referred by this object

CSL_TmrRegsOvly CSL_TmrObj::regs
Pointer to the register overlay structure of the timer

21.3.2 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
imer 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

21.3.3 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.

21.3.4 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.

21.3.5 CSL_TmrHwSetup

Detailed Description
Hardware setup structure.

Field Documentation

CSL_TmrClksrc CSL_TmrHwSetup::tmrClksrcHi
CLKSRC determines the selected clock source for the timer

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::tmrInvInpHi
Timer input inverter control. Only affects operation if CLKSRC=1, Timer Input pin

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::tmrIpGateHi
TIEN determines if the timer clock is gated by the timer input. Applicable only when CLKSRC=0

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 GP 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

21.3.6 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

21.4 Enumerations

This section lists the enumerations available in the TIMER module.

21.4.1 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:
None

CSL_TMR_CMD_STOP64 — Stop the timer of GPtimer64 OR Chained.
Parameters:
CSL_TmrEnamode

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

21.4.2 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

21.4.3 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

21.4.4 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

21.4.5 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

21.4.6 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

21.4.7 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

21.4.8 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

21.4.9 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

21.4.10 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

21.4.11 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

21.4.12 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

21.4.13 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

21.5 Macros

#define CSL_TMR_CONFIG_DEFAULTS
Value:

```
{ \
    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
Value:

```
{ \
    CSL_TMR_PRDLO_RESETVAL, \
    CSL_TMR_PRDHI_RESETVAL, \
    CSL_TMR_TIMLO_RESETVAL, \
    CSL_TMR_TIMHI_RESETVAL, \
    (CSL_TmrIpGate)CSL_TMR_TCR_TIEN_HI_RESETVAL, \
    (CSL_TmrClksrc)CSL_TMR_TCR_CLKSRC_HI_RESETVAL, \
    (CSL_TmrPulseWidth)CSL_TMR_TCR_PWID_HI_RESETVAL, \
    (CSL_TmrClockPulse)CSL_TMR_TCR_CP_HI_RESETVAL, \
    (CSL_TmrInvInp)CSL_TMR_TCR_INVINP_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.

21.6 Typedefs

typedef CSL_TmrObj* CSL_TmrHandle
This is a pointer to CSL_ TmrObj & is passed as the first parameter to all TIMER CSL APIs

Chapter 22
TSIP MODULE

Topics

22.1 Overview

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

The TSIP is a serial interface peripheral with timeslot data management and an integrated DMA capability. The primary purpose of this peripheral is to provide a glueless interface to common telecom serial data streams and efficient internal routing of the data to designated memories in a multi-CPU device.

The TSIP provides 8 serial transmit pins and 8 serial receive pins that connect directly to TEMUX devices. Internally the TSIP offers multiple channels of timeslot data management and multi-channel DMA capability that allow individual timeslots to be selectively processed.

The 3 TSIP's are controlled by 3 different LPSC's. TSIP0 is controlled by LPSC10. TSIP1 is controlled by LPSC11. TSIP2 is controlled by LPSC12. This is done so that TSIP's can be independently clock gated.

22.2 Functions

This section lists the Functions available in the TSIP module.

22.2.1 CSL_tsipInit

CSL_Status CSL_tsipInit (CSL_TsipContext * pContext)

Description
This is the initialization function for the TSIP 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 TSIP 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 TSIP is initialized

Modifies
None

Example

```
CSL_Status status;
...

status = CSL_tsipInit(NULL);
```

22.2.2 CSL_tsipOpen

```
CSL_TsipHandle CSL_tsipOpen (
                              CSL_TsipObj *       pTsipObj,
                              CSL_InstNum         tsipNum,
                              CSL_TsipParam *     pTsipParam,
                              CSL_Status *        pStatus
                              )
```

Description
This function returns the handle to the HPI controller instance. This handle is passed to all other CSL APIs.

Arguments

pTsipObj	Pointer to the object that holds reference to the instance of HPI requested after the call. Memory for this object should be allocated by the user
tsipNum	Instance of TSIP to be opened
pTsipParam	Module specific parameters.
pStatus	Pointer to the variable that holds the status of the open call

Return Value
`CSL_TsipHandle`

- `CSL_TsipHandle` to the requested instance of TSIP if the call is successful, otherwise NULL is returned.

Pre Condition
CSL_tsipInit() must be called before calling this function. Memory for the CSL_TsipObj must be allocated outside this call. This object must be retained while using this peripheral instance.

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

- `CSL_SOK` - Valid TSIP handle is returned
- `CSL_ESYS_FAIL` - The TSIP instance is invalid
- `CSL_ESYS_INVPARAMS` - Invalid parameter

2. TSIP object structure is populated

Modifies
1. The status variable
2. object structure

Example

```
CSL_TsipHandle    hTsip;
CSL_TsipObj       tsipObj;
CSL_Status        status;
...
hTsip = CSL_tsipOpen(&tsipObj, CSL_TSIP_0, NULL, &status);
...
```

22.2.3 CSL_tsipClose

CSL_Status CSL_tsipClose (CSL_TsipHandle hTsip)

Description
Unreserves the TSIP identified by the handle passed.

Arguments

hTsip	Handle to the TSIP

Return Value
CSL_Status

- CSL_SOK - Close Successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipClose().

Post Condition
None

Modifies
TSIP Handle

Example

```
CSL_Status       status;
CSL_tsipHandle   hTsip;
...

status = CSL_tsipClose(hTsip);
...
```

22.2.4 CSL_tsipChHwSetup

CSL_Status CSL_tsipChHwSetup (CSL_TsipHandle hTsip,
CSL_TsipHwSetup * setup
)

Description
Configures the TSIP using the values passed in the setup structure.

Arguments

```
hTsip       TSIP handle returned by successful 'open'

setup       Pointer to the initialized setup structure
```

Return Value
CSL_Status

- CSL_SOK - Tsip Channel setup Successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipChHwSetup().

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

Modifies
TSIP registers

Example
```
CSL_tsipHandle    hTsip;
CSL_TsipHwSetup hwSetup = CSL_TSIP_HWSETUP_DEFAULTS;
...

// Init Successfully done
...
// Open Successfully done
...
CSL_tsipChHwSetup(hTsip, &hwSetup);
...
```

22.2.5 CSL_tsipGetHwSetup

CSL_Status CSL_tsipGetHwSetup (CSL_TsipHandle **hTsip,**
CSL_TsipHwSetup * **hwSetup**
)

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

Arguments

hTsip	TSIP handle returned by successful 'open'
hwSetup	Pointer to @a CSL_TsipHwSetup structure to read the setup parameters

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_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipGetHwSetup().

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

Modifies
hwSetup structure

Example
```
CSL_TsipHandle          hTsip;
CSL_Status              status;

CSL_TsipGblSetup        gblSetup;
```

```
CSL_TsipFrameSetup      frameSetup;
CSL_TsipClkSetup        clkSetup;
CSL_TsipEmu             emuMode;
CSL_TsipHwSetup         readSetup;
...
status = CSL_tsipGetHwSetup(hTsip, &readSetup);
```

22.2.6 CSL_tsipGetHwStatus

CSL_Status CSL_tsipGetHwStatus (CSL_TsipHandle hTsip,
CSL_TsipHwStatusQuery query,
void * response
)

Description
Gets the status of different operations or some setup-parameters of TSIP. The status is returned through the third parameter.

Arguments

```
hTsip       TSIP handle returned by successful 'open'

query       The query to this API of TSIP which indicates the
            status to be returned.

            Query command, refer @a CSL_TsipHwStatusQuery for the
            list of query commands supported

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 - Status info return successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVQUERY - Invalid query command

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipGetHwStatus(). Refer to CSL_TsipHwStatusQuery for the argument to be passed along with the corresponding query command.

Post Condition
None

Modifies
Third parameter, response value

Example

```
CSL_TsipHandle    hTsip;
CSL_Status        status;
CSL_TsipPid       response;
...
```

```
status = CSL_tsipGetHwStatus( hTsip,
                              CSL_TSIP_QUERY_PID,
                              &response);
...
```

22.2.7 CSL_tsipHwControl

CSL_Status CSL_tsipHwControl (CSL_TsipHandle **hTsip,**
CSL_TsipControlCmd **cmd,**
void * arg
)

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of TSIP.

Arguments

hTsip	TSIP handle returned by successful 'open'
cmd	The command to this API indicates the action to be taken on TSIP. Control command, refer @a CSL_TsipControlCmd for the list of commands supported
arg	An optional argument. Optional argument as per the control command, @a void * casted

Return Value
CSL_Status

- CSL_SOK - Command successful.
- CSL_ESYS_BADHANDLE - Invalid handle.
- CSL_ESYS_INVCMD - Invalid command.

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipHwControl().
Refer to CSL_TsipHwControlCmd for the argument type (void*) that needs to be passed with the control command

Post Condition
None

Modifies
The hardware registers of TSIP.

Example

```
CSL_Status        status;
CSL_BitMask16     ctrlMask;
CSL_TsipHandle    hTsip;
...
// TSIP object defined and HwSetup structure defined and
initialized
```

```
...

// Init successfully done
...
// Open successfully done
...
// HwSetup successfully done
...

ctrlMask = 1;
status = CSL_tsipHwControl(hTsip,
                           CSL_TSIP_CMD_SET_PRI,
                           &ctrlMask);
```

22.2.8 CSL_tsipHwSetup

CSL_Status CSL_tsipHwSetup (CSL_TsipHandle **hTsip,**
CSL_TsipHwSetup * **setup**
)

Description
Configures the TSIP using the values passed in the setup structure.

Arguments

```
hTsip       TSIP handle returned by successful 'open'

setup       Pointer to the initialized setup structure
```

Return Value
CSL_Status

- CSL_SOK - HwSetup Successful
- CSL_ESYS_BADHANDLE - Invalid handle
- CSL_ESYS_INVPARAMS - Invalid parameter

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling CSL_tsipHwSetup(). The setup structure consists of the logically grouped sub-structures. The main setup structure as well as these sub-structures must be filled up before calling this function.

Post Condition
The registers of the specified TSIP instance will be setup according to value passed.

Modifies
Hardware registers of the specified TSIP instance.

Example

```
CSL_tsipHandle    hTsip;
CSL_TsipHwSetup   hwSetup = CSL_TSIP_HWSETUP_DEFAULTS;
...

// Init Successfully done
...
// Open Successfully done
```

```
...
CSL_tsipHwSetup(hTsip, &hwSetup);
...
```

22.2.9 _CSL_tsipCfgTimeslot

void _CSL_tsipCfgTimeslot (CSL_TsipHandle **hTsip,**
CSL_TsipTimeslotCfg * **slot**
)

Description
Function to configure a time slot for a particular data format

Arguments

```
hTsip       Handle to TSIP Obj used

slot        time slot configuration for TSIP
```

Return Value
None

Pre Condition
Both CSL_tsipInit() and CSL_tsipOpen() must be called successfully in order before calling this function.

Post Condition
TSIP registers are configured according to the hardware setup parameters

Modifies
TSIP registers will be setup according to value passed

Example

```
CSL_TsipObj          tsipObj;
CSL_TsipHandle       hTsip;
CSL_Status           status;
CSL_TsipTimeslotCfg  slot;
...

hTsip = CSL_tsipOpen(&tsipObj, CSL_TSIP_1, NULL, &status);
...
_CSL_tsipCfgTimeslot(hTsip, slot);
...
```

22.2.10 CSL_tsipGetBaseAddress

CSL_Status CSL_tsipGetBaseAddress (
CSL_InstNum tsipNum,
CSL_TsipParam * **pTsipParam,**
CSL_TsipBaseAddress * **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_tsipOpen()

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. MMR's go to an alternate location.

Arguments

tsipNum	Specifies the instance of the TSIP to be opened.
pTsipParam	Module specific parameters.
pBaseAddress	Pointer to baseaddress structure containing base address details.

Return Value

CSL_Status

- CSL_SOK - Open call is successful
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Inavlid parameters

Pre Condition

None

Post Condition

Base Address structure is populated

Modifies

Base address structure is modified.

Example

```
CSL_Status              status;
CSL_TsipBaseAddress     baseAddress;

...
status = CSL_tsipGetBaseAddress(    CSL_TSIP_1,
                                    NULL,
                                    &baseAddress);
```

22.3 Data Structures

22.3.1 CSL_TsipBaseAddress

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

Field Documentation

CSL_TsipRegsOvly CSL_TsipBaseAddress::regs
Base-address of the Configuration registers of TSIP.

22.3.2 CSL_TsipChanSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters of a channel configuration.

Field Documentation

Uint32 CSL_TsipChanSetup::baseaddr
Memory base Address

Uint32* CSL_TsipChanSetup::bitmap
Assign a filled up bitmap

Uint32 CSL_TsipChanSetup::falloc
Frame Allocation Register

Uint32 CSL_TsipChanSetup::fcount
Frame Count Register

Uint32 CSL_TsipChanSetup::fsize
Frame Size Register

22.3.3 CSL_TsipChanstat

Detailed Description
This structure is used for obtaining the status of a channel.

Field Documentation

CSL_TsipChanum CSL_TsipChanstat::chanum
Tsip channel number

CSL_TsipChst CSL_TsipChanstat::stat
Tsip channel status

22.3.4 CSL_TsipContext

Detailed Description
TSIP specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_TsipContext::contextInfo
Context information of TSIP. The below declaration is just a place-holder for future implementation.

22.3.5 CSL_TsipErrInfo

Detailed Description
This structure is used for obtaining the error information for a particular channel. Both the receive and transmit errors are logged in the same fifo.

Field Documentation

CSL_TsipChanum CSL_TsipErrInfo::chanum
Tsip Clock Redundancy selection

Uint32 CSL_TsipErrInfo::errcnt
Tsip Error Count

Uint32 CSL_TsipErrInfo::errcode
Tsip Error Code

Uint32 CSL_TsipErrInfo::errqov
Tsip Error Queue Overflow

Uint32 CSL_TsipErrInfo::info
Tsip Channel Error Queue Register information

22.3.6 CSL_TsipFrameSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

Field Documentation

CSL_TsipFramecount CSL_TsipFrameSetup::fcount
Number of frames in a superframe

CSL_TsipFramesize CSL_TsipFrameSetup::fsize
Frame Size

22.3.7 CSL_TsipGblSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

Field Documentation

CSL_TsipClkd CSL_TsipGblSetup::clkd
Tsip Clock Redundancy selection

CSL_TsipEndian CSL_TsipGblSetup::endian
endian selection

CSL_TsipPri CSL_TsipGblSetup::maxpri
Maximum DMA priority

CSL_TsipPri CSL_TsipGblSetup::pri
DMA priority

22.3.8 CSL_TsipHwSetup

Detailed Description
This is the Setup structure for configuring TSIP using CSL_tsipHwSetup() function.

Field Documentation

Uint32* CSL_TsipHwSetup::ChannelNumber
Rcv Channel Configuration B

void* CSL_TsipHwSetup::extendSetup
Any extra parameters, for future use

CSL_TsipGblSetup* CSL_TsipHwSetup::gbl
Global configuration parameters

CSL_TsipChanSetup* CSL_TsipHwSetup::rchana
Rcv Channel Configuration A

CSL_TsipChanSetup* CSL_TsipHwSetup::rchanb
Rcv Channel Configuration B

CSL_TsipRclkSetup* CSL_TsipHwSetup::rclk
Rcv Clock setup

CSL_TsipFrameSetup* CSL_TsipHwSetup::rframe
Receive Frame Setup

CSL_TsipIntSetup* CSL_TsipHwSetup::rint
Receive Interrupt Setup

CSL_TsipChanSetup* CSL_TsipHwSetup::xchana
Xmt Channel Configuration B

CSL_TsipChanSetup* CSL_TsipHwSetup::xchanb
Xmt Channel Configuration B

CSL_TsipXclkSetup* CSL_TsipHwSetup::xclk
Xmt Clock setup

CSL_TsipFrameSetup* CSL_TsipHwSetup::xframe
Transmit Frame Setup

CSL_TsipIntSetup* CSL_TsipHwSetup::xint
Receive Interrupt Setup

22.3.9 CSL_TsipIntSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

Field Documentation

CSL_TsipInt CSL_TsipIntSetup::fdelay
Delay for the interrupt

CSL_TsipInt CSL_TsipIntSetup::frint
Interrupt selection for Frame Transfer

CSL_TsipInt CSL_TsipIntSetup::sfint
Interrupt selection for Super frame

22.3.10 CSL_TsipObj

Detailed Description
This structure/object holds the context of the instance of TSIP opened using CSL_tsipOpen() function.
Pointer to this object is passed as TSIP Handle to all TSIP CSL APIs. CSL_tsipOpen() function initializes this structure based on the parameters passed

Field Documentation

CSL_InstNum CSL_TsipObj::perNum
Instance of TSIP being referred by this object

CSL_TsipRegsOvly CSL_TsipObj::regs
Pointer to the register overlay structure of the TSIP

22.3.11 CSL_TsipParam

Detailed Description
TSIP specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_TsipParam::flags

Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

22.3.12 CSL_TsipPerId

Detailed Description
Pointer to this structure is used as the third argument in CSL_tsipGetHwStatus () for getting revision, type and class info of TSIP.

Field Documentation

Uint16 CSL_TsipPerId::major
Major Revision

Uint16 CSL_TsipPerId::minor
Minor Revision

Uint16 CSL_TsipPerId::modid
Module Identifier of the peripheral

22.3.13 CSL_TsipRclkSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring Receive Clock and Frame Sync generation parameters.

Field Documentation

CSL_TsipClkm CSL_TsipRclkSetup::clkmode
Receive Clock Mode

CSL_TsipClkSrc CSL_TsipRclkSetup::clksrc
RCV frame sync mode

Uint32 CSL_TsipRclkSetup::datd
Data delay

CSL_TsipClkp CSL_TsipRclkSetup::dclkp
Receive Data Clock Polarity

CSL_TsipDataRate CSL_TsipRclkSetup::drate
Receive Data Rate

CSL_TsipClkp CSL_TsipRclkSetup::fclkp
Receive Frame Clock Polarity

CSL_TsipFsyncp CSL_TsipRclkSetup::fsyncp
Receive Frame sync Polarity

22.3.14 CSL_TsipTimeslotCfg

Detailed Description
Pointer to this structure is used as the third argument in CSL_tsipGetHwStatus () for getting revision, type and class info of TSIP.

Field Documentation

CSL_TsipChanCfg CSL_TsipTimeslotCfg::chanum
Channel number

CSL_TsipTimeslot CSL_TsipTimeslotCfg::slot
Time slot

Uint16 CSL_TsipTimeslotCfg::slotnum
Slot number

22.3.15 CSL_TsipXclkSetup

Detailed Description
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring Transmit Clock and Frame Sync generation parameters.

Field Documentation

CSL_TsipClkm CSL_TsipXclkSetup::clkmode
Transmit Data Rate

CSL_TsipClkSrc CSL_TsipXclkSetup::clksrc
Clock and Frame Sync source

Uint32 CSL_TsipXclkSetup::datd
Data delay

CSL_TsipClkp CSL_TsipXclkSetup::dclkp
Transmit Data Clock Polarity

CSL_TsipDataRate CSL_TsipXclkSetup::drate
Transmit Data Rate

CSL_TsipClkp CSL_TsipXclkSetup::fclkp
Transmit Frame Clock Polarity

CSL_TsipFsyncp CSL_TsipXclkSetup::fsyncp
Transmit Frame sync Polarity

CSL_TsipXmtDis CSL_TsipXclkSetup::outdis
Output disable state

22.4 Enumerations

22.4.1 CSL_TsipChanCfg

enum CSL_TsipChanCfg

Enumeration values:

CSL_TSIP_XMT_CHA0_CFGA	Transmit Channel 0 Config A
CSL_TSIP_XMT_CHA1_CFGA	Transmit Channel 1 Config A
CSL_TSIP_XMT_CHA2_CFGA	Transmit Channel 2 Config A
CSL_TSIP_XMT_CHA3_CFGA	Transmit Channel 3 Config A
CSL_TSIP_XMT_CHA4_CFGA	Transmit Channel 4 Config A
CSL_TSIP_XMT_CHA5_CFGA	Transmit Channel 5 Config A
CSL_TSIP_RCV_CHA0_CFGA	Receive Channel 0 Config A
CSL_TSIP_RCV_CHA1_CFGA	Receive Channel 1 Config A
CSL_TSIP_RCV_CHA2_CFGA	Receive Channel 2 Config A
CSL_TSIP_RCV_CHA3_CFGA	Receive Channel 3 Config A
CSL_TSIP_RCV_CHA4_CFGA	Receive Channel 4 Config A
CSL_TSIP_RCV_CHA5_CFGA	Receive Channel 5 Config A
CSL_TSIP_XMT_CHA0_CFGB	Receive Channel 0 Config A
CSL_TSIP_XMT_CHA1_CFGB	Receive Channel 1 Config A
CSL_TSIP_XMT_CHA2_CFGB	Receive Channel 2 Config A
CSL_TSIP_XMT_CHA3_CFGB	Receive Channel 3 Config A
CSL_TSIP_XMT_CHA4_CFGB	Receive Channel 4 Config A
CSL_TSIP_XMT_CHA5_CFGB	Receive Channel 5 Config A
CSL_TSIP_RCV_CHA0_CFGB	Receive Channel 0 Config A
CSL_TSIP_RCV_CHA1_CFGB	Receive Channel 1 Config A
CSL_TSIP_RCV_CHA2_CFGB	Receive Channel 2 Config A
CSL_TSIP_RCV_CHA3_CFGB	Receive Channel 3 Config A
CSL_TSIP_RCV_CHA4_CFGB	Receive Channel 4 Config A

CSL_TSIP_RCV_CHA5_CFGB	Receive Channel 5 Config A

22.4.2 CSL_TsipChanum

enum CSL_TsipChanum

Enumeration values:

CSL_TSIP_XMT_CHA0	Transmit Channel 0
CSL_TSIP_XMT_CHA1	Transmit Channel 1
CSL_TSIP_XMT_CHA2	Transmit Channel 2
CSL_TSIP_XMT_CHA3	Transmit Channel 3
CSL_TSIP_XMT_CHA4	Transmit Channel 4
CSL_TSIP_XMT_CHA5	Transmit Channel 5
CSL_TSIP_RCV_CHA0	Receive Channel 0
CSL_TSIP_RCV_CHA1	Receive Channel 1
CSL_TSIP_RCV_CHA2	Receive Channel 2
CSL_TSIP_RCV_CHA3	Receive Channel 3
CSL_TSIP_RCV_CHA4	Receive Channel 4
CSL_TSIP_RCV_CHA5	Receive Channel 5

22.4.3 CSL_TsipChst

enum CSL_TsipChst

Enumeration values:

CSL_TSIP_CHST_INACTIVE	Channel is Inactive
CSL_TSIP_CHST_AACTIVE	A Config/Bitmap is used
CSL_TSIP_CHST_BACTIVE	B Config/Bitmap is used

22.4.4 CSL_TsipClkd

enum CSL_TsipClkd

Enumeration values:

CSL_TSIP_CLKD_REDUN	Redundant Clock Mode
CSL_TSIP_CLKD_DUAL	Dual Clock Mode

22.4.5 CSL_TsipClkm

enum CSL_TsipClkm

Enumeration values:

CSL_TSIP_CLKM_DBL	Double rate clock
CSL_TSIP_CLKM_SGL	Single rate clock

22.4.6 CSL_TsipClkp

enum CSL_TsipClkp

Enumeration values:

CSL_TSIP_CLKP_RISING	Rising Polarity
CSL_TSIP_CLKP_FALLING	Falling Polarity

22.4.7 CSL_TsipClkSrc

enum CSL_TsipClkSrc

Enumeration values:

CSL_TSIP_CLKSRC_A	Clock and Frame sync Source A
CSL_TSIP_CLKSRC_B	Clock and Frame sync Source B

22.4.8 CSL_TsipControlCmd

enum CSL_TsipControlCmd
This is the set of control commands that are passed to CSL_tsipHwControl(), 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_TSIP_CMD_SET_XCLK_SRC	Setup Transmit Clock and Frame Sync Source **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XCLK_MODE	Setup Transmit Clock Mode...Double vs Single Clock **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XDATA_RATE	Setup Transmit Data rate 8/16/32 **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XDIS_OUTPUT	Setup Transmit Output when transmit is disabled **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XDATA_DELAY	Setup Data Delay

	Parameters:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XDCLK_POL	Setup Transmit data clock polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XFCLK_POL	Setup Receive Frame clock polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_XFSYNC_POL	Setup Transmit Frame Sync Polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RCLK_SRC	Setup Receive Clock Source **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RCLK_MODE	Setup Receive Clock Mode **Parameters**:None **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RDATA_RATE	Setup Receive Data Rate **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RDATA_DELAY	Setup Receive Data Delay **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RDCLK_POL	Setup Receive Data Clock Polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RFCLK_POL	Setup Receive Frame Clock Polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_RFSYNC_POL	Setup Receive Frame Sync Polarity **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_CLKD	Setup Clock redundancy selection **Parameters**:*None* Returns:CSL_SOK
CSL_TSIP_CMD_RST_SIU	Reset SIU **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_ENA_SIU_RCV	Enable SIU Receive **Parameters**:*None* **Returns**:CSL_SOK

CSL_TSIP_CMD_ENA_SIU_XMT	Enable SIU Transmit **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_DIS_SIU_RCV	Disable SIU Receive **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_DIS_SIU_XMT	Disable SIU Transmit **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_PRI	Set the Priority for Dma **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_MAXPRI	Set the maximum priority for the Priority to be set **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_CLR_QOV	Clear the Error Queue Overflow flag **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_CLRQ	Clear the Error Queue **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_POPQ	Pop the top from the Error Queue **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_ENA_CHAN	Enable a Transmit or Receive Channel **Parameters**:None **Returns**:CSL_SOK
CSL_TSIP_CMD_DIS_CHAN	Disable a Transmit or Receive Channel **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_CFG_TIMESLOT	Configure a timeslot in the bitmap **Parameters**:*(CSL_TsipTslotCfg *)* **Returns**:CSL_SOK
CSL_TSIP_CMD_ENA_DMA	Enable DMATCU **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_DIS_DMA	Disable DMATCU **Parameters**:*None* **Returns**:CSL_SOK
CSL_TSIP_CMD_SET_ENDIAN	Setup the word Endianiness for incoming data **Parameters**:*None*

	Returns:CSL_SOK
CSL_TSIP_CMD_RST_DMA	Enable SIU Transmit **Returns**:CSL_SOK

22.4.9 CSL_TsipDataRate

enum CSL_TsipDataRate

Enumeration values:

CSL_TSIP_DATARATE_8M	Data rate is 8 Mbps
CSL_TSIP_DATARATE_16M	Data rate is 16 Mbps
CSL_TSIP_DATARATE_32M	Data rate is 32 Mbps

22.4.10 CSL_TsipEmu

enum CSL_TsipEmu

Enumeration values:

CSL_TSIP_EMU_STOP	Emulation mode stop
CSL_TSIP_EMU_TX_STOP	Emulation mode TX stop
CSL_TSIP_EMU_FREERUN	Emulation free run mode

22.4.11 CSL_TsipEndian

enum CSL_TsipEndian

Enumeration values:

CSL_TSIP_ENDIAN_LITTLE	Little Endian
CSL_TSIP_ENDIAN_BIG	Big Endian

22.4.12 CSL_TsipFramecount

enum CSL_TsipFramecount

Enumeration values:

CSL_TSIP_FRAMECOUNT_1	Frame Count is 0
CSL_TSIP_FRAMECOUNT_256	Frame Count is 255

22.4.13 CSL_TsipFramesize

enum CSL_TsipFramesize

Enumeration values:

CSL_TSIP_FRAMESIZE_1	Frame Count is 0

CSL_TSIP_FRAMESIZE_128	Frame Count is 255

22.4.14 CSL_TsipFsyncp

enum CSL_TsipFsyncp

Enumeration values:

CSL_TSIP_FSYNCP_ALOW	Active Low
CSL_TSIP_FSYNCP_AHIGH	Active High

22.4.15 CSL_TsipHwStatusQuery

enum CSL_TsipHwStatusQuery
This is the set of query commands to get the status of various operations in TSIP
The arguments, if any, to be passed with each command are specified next to that command.

Enumeration values:

CSL_TSIP_QUERY_PID	Queries the current XMT block **Parameters**:*(CSL_TsipPeriId *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_XFRFC	Queries the current XMT block **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_RFRFC	Queries the current XMT block **Parameters**:*(Uint32 *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_BMST	Queries the current status of the bitmap **Parameters**:*(CSL_TsipChst *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_CHST	Queries the current status of a channel **Parameters**:*(CSL_TsipChst *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_CHANCFG	Queries the current Channel Configuration parameters **Parameters**:*(CSL_TsipChan *)* **Returns**:CSL_SOK
CSL_TSIP_QUERY_ERR_INFO	Queries the Error Information for a channel **Parameters**:*(CSL_TsipErrInfo *)* **Returns**:CSL_SOK s

22.4.16 CSL_TsipInt

enum CSL_TsipInt

Enumeration values:

CSL_TSIP_INT_ACK	Interrupt on Acknowledgement
CSL_TSIP_INT_DLY	Interrupt on Delay
CSL_TSIP_INT_ACKORDLY	Interrupt on either Acknowledgement or Delay
CSL_TSIP_INT_ACKNDLY	Interrupt on both Acknowledgement and Delay

22.4.17 CSL_TsipPri

enum CSL_TsipPri

Enumeration values:

CSL_TSIP_PRI_0	Priority 0
CSL_TSIP_PRI_1	Priority 1
CSL_TSIP_PRI_2	Priority 2
CSL_TSIP_PRI_3	Priority 3
CSL_TSIP_PRI_4	Priority 4
CSL_TSIP_PRI_5	Priority 5
CSL_TSIP_PRI_6	Priority 6
CSL_TSIP_PRI_7	Priority 7

22.4.18 CSL_TsipTimeslot

enum CSL_TsipTimeslot

Enumeration values:

CSL_TSIP_TIMESLOT_DISABLED	Timeslot Disabled
CSL_TSIP_TIMESLOT_LINEAR	Linear
CSL_TSIP_TIMESLOT_ULAW	Linear
CSL_TSIP_TIMESLOT_ALAW	Linear

22.4.19 CSL_TsipXmtDis

enum CSL_TsipXmtDis

Enumeration values:

CSL_TSIP_XMTDIS_HIGHIMP	High Impedance
CSL_TSIP_XMTDIS_LOW	Driven Low
CSL_TSIP_XMTDIS_HIGH	Driven HIGH

22.4.20 CSL_TsipXmtRcv

enum CSL_TsipXmtRcv

Enumeration values:

CSL_TSIP_XMT_CHAN	Transmit Channel
CSL_TSIP_RCV_CHAN	Receive Channel

22.5 Macros

#define CSL_ETSIP_INVCNTLCMD (CSL_ETSIP_FIRST - 0)
Invalid Control Command

#define CSL_ETSIP_INVMODE (CSL_ETSIP_FIRST - 5)
Invalid mode to conduct operation

#define CSL_ETSIP_INVPARAMS (CSL_ETSIP_FIRST - 2)
Invalid Parameter

#define CSL_ETSIP_INVQUERY (CSL_ETSIP_FIRST - 1)
Invalid Query

#define CSL_ETSIP_INVSIZE (CSL_ETSIP_FIRST - 3)
Invalid Size

#define CSL_ETSIP_NOTEXIST (CSL_ETSIP_FIRST - 4)
'Does not exist'

#define CSL_TSIP_GBLSETUP_DEFAULTS
Value:
```
{ \
      CSL_TSIP_CLKD_REDUN,   \
      CSL_TSIP_ENDIAN_LITTLE, \
      CSL_TSIP_PRI_0, \
      CSL_TSIP_PRI_7  \
}
```
Global setup default values

#define CSL_TSIP_RCLK_SETUP_DEFAULTS
Value:
```
{ \
      CSL_TSIP_CLKSRC_A, \
      0, \
      CSL_TSIP_FSYNCP_ALOW, \
      CSL_TSIP_CLKP_RISING, \
      CSL_TSIP_CLKP_RISING, \
      CSL_TSIP_DATARATE_8M, \
      CSL_TSIP_CLKM_DBL \
}
```
Receive clock setup default values

#define CSL_TSIP_XCLK_SETUP_DEFAULTS
Value:
```
{ \
      CSL_TSIP_CLKSRC_A, \
      0, \
      CSL_TSIP_XMTDIS_HIGHIMP, \
      CSL_TSIP_FSYNCP_ALOW, \
      CSL_TSIP_CLKP_RISING, \
      CSL_TSIP_CLKP_RISING, \
      CSL_TSIP_DATARATE_8M, \
```

```
      CSL_TSIP_CLKM_DBL \
}
```
Transmit clock setup default values

```
#define CSL_TSIP_HWSETUP_DEFAULTS { \
CSL_TSIP_GBLSETUP_DEFAULTS, \
CSL_TSIP_XCLK_SETUP_DEFAULTS, \
CSL_TSIP_RCLK_SETUP_DEFAULTS, \
{ CSL_TSIP_FRAMECOUNT_1, CSL_TSIP_FRAMESIZE_1 }, \
{ CSL_TSIP_FRAMECOUNT_1, CSL_TSIP_FRAMESIZE_1 }, \
{ CSL_TSIP_INT_ACK, CSL_TSIP_INT_ACK, CSL_TSIP_INT_ACK }, \
{ CSL_TSIP_INT_ACK, CSL_TSIP_INT_ACK, CSL_TSIP_INT_ACK }, \
      { /* TX-Ch-A */ \
      0x00000000, /* Memory base Address */ \
      0x0,  /* Frame Allocation Register */ \
      0x0,  /* Frame Size Register */ \
      0x0, /* Frame Count Register */ \
      0x0 /* Assign a filled up bitmap */ \
    }, \
    { /* TX-Ch-B */ \
      0x00000000, /* Memory base Address */ \
      0x0,  /* Frame Allocation Register */ \
      0x0,  /* Frame Size Register */ \
      0x0, /* Frame Count Register */ \
      0x0 /* Assign a filled up bitmap */ \
    }, \
    { /* RX-Ch-A */ \
      0x00000000, /* Memory base Address */ \
      0x0,  /* Frame Allocation Register */ \
      0x0,  /* Frame Size Register */ \
      0x0, /* Frame Count Register */ \
      0x0 /* Assign a filled up bitmap */ \
    }, \
    { /* RX-Ch-B */ \
      0x00000000, /* Memory base Address */ \
      0x0,  /* Frame Allocation Register */ \
      0x0,  /* Frame Size Register */ \
      0x0, /* Frame Count Register */ \
      0x0 /* Assign a filled up bitmap */ \
    }, \
    0x0,  /* Rcv Channel Configuration B*/ \
    0x0 \
}
```
TSIP hardware setup default values

22.6 Typedefs

typedef CSL_TsipPerId CSL_TsipPerId
Pointer to this structure is used as the third argument in CSL_tsipGetHwStatus () for getting revision, type and class info of TSIP.

typedef CSL_TsipTimeslotCfg CSL_TsipTimeslotCfg
Pointer to this structure is used as the third argument in CSL_tsipGetHwStatus () for getting revision, type and class info of TSIP.

typedef CSL_TsipXclkSetup CSL_TsipXclkSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring Transmit Clock and Frame Sync generation parameters.

typedef CSL_TsipRclkSetup CSL_TsipRclkSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring Receive Clock and Frame Sync generation parameters.

typedef CSL_TsipGblSetup CSL_TsipGblSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

typedef CSL_TsipErrInfo CSL_TsipErrInfo
This structure is used for obtaining the error information for a particular channel. Both the receive and transmit errors are logged in the same fifo.

typedef CSL_TsipChanstat CSL_TsipChanstat
This structure is used for obtaining the status of a channel.

typedef CSL_TsipFrameSetup CSL_TsipFrameSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

typedef CSL_TsipIntSetup CSL_TsipIntSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters global to TSIP.

typedef CSL_TsipChanSetup CSL_TsipChanSetup
This is a sub-structure in CSL_TsipHwSetup. This structure is used for configuring the parameters of a channel configuration.

typedef CSL_TsipHwSetup CSL_TsipHwSetup
This is the Setup structure for configuring TSIP using CSL_tsipHwSetup() function.

typedef struct CSL_TsipObj CSL_TsipObj
This structure/object holds the context of the instance of TSIP opened using CSL_tsipOpen() function.
Pointer to this object is passed as TSIP Handle to all TSIP CSL APIs. CSL_tsipOpen() function initializes this structure based on the parameters passed

typedef CSL_TsipObj * CSL_TsipHandle
This is a pointer to CSL_TsipObj and is passed as the first parameter to all TSIP CSL APIs.

Chapter 23
UTOPIA2 MODULE

Topics

23.1 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/16-bit Slave-only interface. Utopia DMA transfers are being serviced by UTOPIA-PDMA. 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.

The UTOPIA slave consists of the transmit interface and the receive interface.

23.2 Functions

This section lists the Functions available in the UTOPIA2 module.

23.2.1 CSL_utopia2Init

CSL_Status CSL_utopia2Init (CSL_Utopia2Context * pContext)

Description
This is the initialization function for the UTOPIA2 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 UTOPIA2 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 UTOPIA2 is initialized

Modifies
None

Example

```
CSL_Status status;
...
status = CSL_utopia2Init(NULL);
...
```

23.2.2 CSL_utopia2Open

```
CSL_Utopia2Handle CSL_utopia2Open (   CSL_Utopia2Obj     *pUtopia2Obj,
                                      CSL_InstNum        utopia2Num,
                                      pUtopia2Param      *pUtopia2Param,
                                      CSL_Status         * pStatus
                                  )
```

Description
This function reserves the specified UTOPIA2 for use. The device can be re-opened anytime after it has been normally closed, if so required. The UTOPIA2 handle returned by this call is input as an essential argument for the rest of the APIs in UTOPIA2 module. If hwSetup parameter is non-

NULL, it configures UTOPIA2 depending on the parameters passed in this structure, after successful open

Arguments

pUtopia2Obj	Pointer to UTOPIA2 object that holds the context.Memory for this object should be allocated by the User
utopia2Num	Instance of UTOPIA2 to which a handle is requested. There is only one instance of the utopia2 available. So, the value for this parameter will be CSL_UTOPIA2 always
pUtopia2Param	Pointer to module specific parameters
pStatus	This returns the status (success/error) of the call. The user may pass 'NULL', if status information is not required

Return Value
CSL_Utopia2Handle

- Valid UTOPIA2 handle will be returned if status value is equal to CSL_SOK.

Pre Condition
The UTOPIA2 must be successfully initialized via CSL_utopia2Init() before calling this function.

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

CSL_SOK - Valid UTOPIA2 handle is returned
CSL_ESYS_FAIL - The UTOPIA2 instance is invalid
CSL_ESYS_INVPARAMS - Invalid parameter

2. UTOPIA2 object structure is populated

Modifies
1. The status variable
2. UTOPIA2 object structure

Example

```
CSL_Status           status;
CSL_Utopia2Obj       utopia2Obj;
CSL_Utopia2Handle    hUtopia2;
...
hUtopia2 = CSL_utopia2Open(&utopia2Obj, CSL_UTOPIA2, NULL,
                    &status);
```

23.2.3 CSL_utopia2Close

CSL_Status CSL_utopia2Close (CSL_Utopia2Handle hUtopia2)

Description
This function closes the specified instance of UTOPIA2.

Arguments

```
hUtopia2    Pointer to the object that holds reference to the
            instance of Utopia2 requested after the call
```

Return Value
CSL_Status

- CSL_SOK - Close successful
- CSL_ESYS_BADHANDLE - Invalid handle

Pre Condition
The UTOPIA2 instance should be opened before this close operation.

Post Condition
The UTOPIA2 CSL APIs can not be called until the UTOPIA2 CSL is reopened again using CSL_utopia2Open().

Modifies
None

Example

```
CSL_Utopia2Handle       hUtopia2;
CSL_Status              status;
...
status = CSL_utopia2Close(hUtopia2);
...
```

23.2.4 CSL_utopia2GetHwSetup

CSL_Status CSL_utopia2GetHwSetup (CSL_Utopia2Handle hUtopia2, CSL_Utopia2HwSetup * hwSetup)

Description
It retrieves the hardware setup parameters of the Utopia2 specified by the given handle.

Arguments

```
hUtopia2    Handle to the utopia2

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
CSL_utopia2Init() and CSL_utopia2Open() must be called successfully in order before calling CSL_utopia2GetHwSetup().

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

Modifies
hwSetup variable

Example

```
CSL_Utopia2Handle   hUtopia2;
CSL_Utopia2HwSetup  hwSetup;
...
status = CSL_utopia2GetHwSetup(hUtopia2, &hwSetup);
...
```

23.2.5 CSL_utopia2GetHwStatus

CSL_Status CSL_utopia2GetHwStatus (CSL_Utopia2Handle **hUtopia2,**
CSL_Utopia2HwStatusQuery **utopia2Query,**
void * response
)

Description
Gets the status of the different operations of UTOPIA2.

Arguments

hUtopia2	Handle to the UTOPIA2 instance
utopia2Query	The query to this API of UTOPIA2 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_utopia2Init() and CSL_utopia2Open() must be called successfully in order before calling CSL_utopia2GetHwStatus().

Post Condition
None

Modifies
Third parameter response value

Example

```
CSL_Utopia2Handle         hUtopia2;
CSL_Utopia2HwStatusQuery  query = CSL_UTOPIA2_QUERY_XMT_STATUS;
Uint32                    response;
...

status = CSL_utopia2GetHwStatus(hUtopia2, query, &response);
...
```

23.2.6 CSL_utopia2HwControl

CSL_Status CSL_utopia2HwControl (	**CSL_Utopia2Handle**	**hUtopia2,**
	CSL_Utopia2ControlCmd	**ctrlCmd,**
	void	***arg**
	)	

Description
This function takes an input control command with an optional argument and accordingly controls the operation/configuration of UTOPIA2.

Arguments

hUtopia2	Handle to the UTOPIA2 instance.
ctrlCmd	The command to this API indicates the action to be taken on UTOPIA2.
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

Pre Condition
CSL_utopia2Init() and CSL_utopia2Open() must be called successfully in order before calling CSL_utopia2HwControl().

Post Condition
UTOPIA2 registers are configured according to the command passed.

Modifies
The hardware registers of UTOPIA2.

Example

```
CSL_Utopia2Handle            hUtopia2;
CSL_Utopia2HwControlCmd      cmd = CSL_UTOPIA2_CMD_XMT_ENABLE;
Uint32                       arg;
...
status = CSL_utopia2HwControl(hUtopia2, cmd, NULL);
...
```

23.2.7 CSL_utopia2HwSetup

CSL_Status CSL_utopia2HwSetup (CSL_Utopia2Handle hUtopia2,
CSL_Utopia2HwSetup *utopia2Setup
)

Description
It configures the Utopia2 registers as per the values passed in the hardware setup structure.

Arguments

```
hUtopia2          Handle to the utopia2

utopia2Setup      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_utopia2Init() and CSL_utopia2Open() must be called successfully in order before calling this function.

Post Condition
UTOPIA2 registers are configured according to the hardware setup parameters.

Modifies
UTOPIA2 registers

Example

```
CSL_Status              status;
CSL_Utopia2HwSetup      hwSetup;
CSL_Utopia2Handle       hUtopia2;
...
hwSetup.utopia2Ctrl = (CSL_Utopia2Ctrl)0x80;
.....
status = CSL_utopia2HwSetup(hUtopia2, &hwSetup);
```

23.2.8 CSL_ utopia2GetBaseAddress

CSL_Status CSL_utopia2GetBaseAddress (CSL_InstNum *utopia2Num*,
CSL_Utopia2Param **pUtopia2Param*,
CSL_Utopia2BaseAddress **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_utopia2Open() 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

utopia2Num	Specifies the instance of the utopia2 to be opened.
pUtopia2Param	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 UTOPIA2.
- 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_Utopia2BaseAddress  baseAddress;
...
status = CSL_utopia2GetBaseAddress(CSL_UTOPIA2, NULL,
                                   &baseAddress);
```

23.3 Data Structures

This section lists Data Structures available in the UTOPIA2 module.

23.3.1 CSL_Utopia2Context

Detailed Description
UTOPIA2 specific context information. Present implementation doesn't have any Context information.

Field Documentation

Uint16 CSL_Utopia2Context::contextInfo
Context information of Utopia2. The below declaration is just a place-holder for future implementation.

23.3.2 CSL_Utopia2Obj

Detailed Description
This structure/object holds the context of the instance of UTOPIA2 opened using CSL_utopia2Open() function. Pointer to this object is passed as UTOPIA2 Handle to all UTOPIA2 CSL APIs. CSL_utopia2Open() function initializes this structure based on the parameters passed.

Field Documentation

CSL_InstNum CSL_Utopia2Obj::perNum
Instance of utopia2 being referred by this object

CSL_Utopia2RegsOvly CSL_Utopia2Obj::regs
Pointer to the register overlay structure of the utopia2

23.3.3 CSL_Utopia2ClkSetup

Detailed Description
Utopia2 Clock Setup Command Structure.

Field Documentation

Uint16 CSL_Utopia2ClkSetup::rxClkCount
RCCNT Receive Clock Count

Uint16 CSL_Utopia2ClkSetup::txClkCount
XCCNT Transmit Clock Count

23.3.4 CSL_Utopia2GlobalGetSetup

Detailed Description
Utopia2 Hardware Setup Structure.

Field Documentation

CSL_Utopia2ClkSetup **CSL_Utopia2GlobalGetSetup::getClkSetup**
Pointer to a Utopia2ClkSetup structure

CSL_Utopia2ErrorSetup CSL_Utopia2GlobalGetSetup::getErrorSetup
Pointer to Error Setup structure

CSL_Utopia2UcrSetup **CSL_Utopia2GlobalGetSetup::getUcrSetup**
Pointer to the Utopia2 Control Register Setup

23.3.5 CSL_Utopia2Rrsr

Detailed Description
Utopia2 Receive routing select Structure.

Field Documentation

Uint8 CSL_Utopia2Rrsr::rbyt
Receive byte

Uint8 CSL_Utopia2Rrsr::rbit
Receive bit

23.3.6 CSL_Utopia2RruSetup

Detailed Description
Utopia2 Receive Routing Unit Command Structure.

Field Documentation
Uint16 CSL_Utopia2RruSetup::rmmrMask0
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask1
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask2
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask3
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask4
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask5
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask6
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMask7
Receive Mask Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch0
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch1
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch2
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch3
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch4
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch5
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch6
Receive Match Value

Uint16 CSL_Utopia2RruSetup::rmmrMatch7
Receive Match Value

CSL_Utopia2Rrsr* CSL_Utopia2RruSetup:: rrsr
Receive routing select structure

23.3.7 CSL_Utopia2HwSetup

Detailed Description
Utopia2 Hardware Setup structure.

Field Documentation
CSL_Utopia2UcrSetup* CSL_Utopia2HwSetup::ucrSetup
Pointer to the Utopia2 Control Register Setup

CSL_Utopia2ClkSetup* CSL_Utopia2HwSetup::clkSetup
Pointer to a Utopia2ClkSetup structure

CSL_Utopia2ErrorSetup* CSL_Utopia2HwSetup::errorSetup
Pointer to Error Setup structure

CSL_Utopia2RruSetup * CSL_Utopia2HwSetup::rruSetup
Pointer to Receive Routing Unit structure

void* CSL_Utopia2HwSetup::extendSetup
VOID For later use

23.3.8 CSL_Utopia2UcrSetup

Detailed Description
Utopia2 'Utopia2 Control Register' Setup Command Structure.

Field Documentation

Uint32 CSL_Utopia2UcrSetup::Endianness
BEND Little Endian or Big Endian

Uint16 CSL_Utopia2UcrSetup::phyModeEnable
MPHY PHY Mode Enable

Uint16 CSL_Utopia2UcrSetup::rxUdc
RUDC Receive User Defined cell

Uint16 CSL_Utopia2UcrSetup::rxUtopia2Enable
UREN UTOPIA2 Receive Enable

Uint16 CSL_Utopia2UcrSetup::slendSlid
Slave ID in Master/MPHY mode

Uint16 CSL_Utopia2UcrSetup::txUdc
XUDC Transmit User defined cell

Uint16 CSL_Utopia2UcrSetup::txUtopia2Enable
UXEN UTOPIA2 Transmit port Enable

Uint16 CSL_Utopia2UcrSetup::utopia2BitMode
U16M 8 bit or 16 bit mode

Uint16 CSL_Utopia2UcrSetup::utopia2PollingMode
UPM Valid only when master mode is enable

23.3.9 CSL_Utopia2Param

Detailed Description
UTOPIA2 specific parameters. Present implementation of UTOPIA2 CSL doesn't have any module specific parameters.

Field Documentation

CSL_BitMask16 CSL_Utopia2Param::flags
Bit mask to be used for module specific parameters. The declaration is just a placeholder for future implementation.

23.3.10 CSL_Utopia2BaseAddress

Detailed Description
This structure contains the base address information for UTOPIA2 peripheral instance.

Field Documentation

CSL_Utopia2RegsOvly CSL_Utopia2BaseAddress::regs
Base address of the Configuration registers of UTOPIA2.

23.4 Enumerations

This section lists the Enumerations available in the UTOPIA2 module.

23.4.1 CSL_Utopia2BitMode

enum CSL_Utopia2BitMode
Utopia2 Bit - Mode Enumeration

Enumeration values

CSL_UTOPIA2_8BIT_MODE	Utopia2 8 Bit Mode Select
CSL_UTOPIA2_16BIT_MODE	Utopia2 16 Bit Mode Select

23.4.2 CSL_Utopia2ControlCmd

enum CSL_Utopia2ControlCmd
Enumeration for commands passed to CSL_utopia2HwControl().
This is used to select the commands to control the operations existing setup of UTOPIA2. The arguments to be passed with each enumeration if any are specified next to the enumeration.

Enumeration values

CSL_UTOPIA2_CMD_ERROR_CLEAR	Command to select Utopia2 Error Clear Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_ERROR_ENABLE	Command to select Utopia2 Error Enable Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_ERROR_DISABLE	Command to select Utopia2 Error Disable Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_INTERUPT_CLEAR	Command to select Utopia2 Interrupt Clear Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_INTERUPT_ENABLE	Command to select Utopia2 Interrupt Enable Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_INTERUPT_DISABLE	Command to select Utopia2 Interrupt Disable Configuration **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_CMD_XMT_ENABLE	Command to enable Utopia2 Transmitter **Parameters:** *(None)*

CSL_UTOPIA2_CMD_XMT_DISABLE	Command to disable Utopia2 Transmitter **Parameters:** *(None)*
CSL_UTOPIA2_CMD_REC_ENABLE	Command to enable Utopia2 Receiver **Parameters:** *(None)*
CSL_UTOPIA2_CMD_REC_DISABLE	Command to disable Utopia2 Receiver **Parameters:** *(None)*
CSL_UTOPIA2_CMD_RESET	Command to Utopia2 Reset **Parameters:** *(None)*
CSL_UTOPIA2_CMD_SET_PHY	Command to set PHY Disable Register **Parameters:** *(Uint32 *)*

23.4.3 CSL_Utopia2HwStatusQuery

enum CSL_Utopia2HwStatusQuery
Enumeration for queries passed to CSL_utopia2GetHwStatus().
This is used to get the status of different operations

Enumeration values

CSL_UTOPIA2_QUERY_ERR_STATUS	Query to get Utopia2 Error Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_ERR_PENDING_STATUS	Query to get Utopia2 Error Pending Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_INTR_STATUS	Query to get Utopia2 Interrupt Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_INTR_PENDING_STATUS	Query to get Utopia2 Interrupt pending Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_XMT_STATUS	Query to get Utopia2 Transmitter Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_REC_STATUS	Query to get Utopia2 Receiver Status **Parameters:** *(CSL_BitMask16 *)*
CSL_UTOPIA2_QUERY_RESET_STATUS	Query to get Utopia2 Reset Status **Parameters:** *(Uint32 *)*
CSL_UTOPIA2_QUERY_TX_CELL_AVAIL_STATUS	Query to get Transmit Cell Available for PHY Register Status

	Parameters: *(Uint32 *)*
CSL_UTOPIA2_QUERY_PHY_STATUS	Query to get PHY Disable Register Status **Parameters:** *(Uint32 *)*

23.4.4 CSL_Utopia2EndianMode

enum CSL_Utopia2EndianMode
Utopia2 Endianness Select enumeration.

Enumeration values

CSL_UTOPIA2_LITTLE_ENDIAN_MODE	Utopia2 Little Endian Mode Select
CSL_UTOPIA2_BIG_ENDIAN_MODE	Utopia2 Big Endian Mode Select

23.4.5 CSL_Utopia2ErrorStatCmd

enum CSL_Utopia2ErrorStatCmd
Utopia2 error status enumeration.

Enumeration values

CSL_UTOPIA2_TX_CLK_PRES_ENABLE	Utopia2 transmit clock present enable setup
CSL_UTOPIA2_TX_CLK_FAIL_ENABLE	Utopia2 transmit clock fail enable setup
CSL_UTOPIA2_RX_CLK_PRES_ENABLE	Utopia2 receive clock present enable setup
CSL_UTOPIA2_RX_CLK_FAIL_ENABLE	Utopia2 receive clock fail enable setup
CSL_UTOPIA2_TX_CLK_PRES_DISABLE	Utopia2 transmit clock present disable setup
CSL_UTOPIA2_TX_CLK_FAIL_DISABLE	Utopia2 transmit clock fail disable setup
CSL_UTOPIA2_RX_CLK_PRES_DISABLE	Utopia2 receive clock present disable setup
CSL_UTOPIA2_RX_CLK_FAIL_DISABLE	Utopia2 receive clock fail disable setup

23.4.6 CSL_Utopia2GetErrorSetup

enum CSL_Utopia2GetErrorSetup
Utopia2 error interrupt status enumeration

Enumeration values

CSL_UTOPIA2_TX_CLK_PRES_INTR_ENABLE	Utopia2 transmit clock present status
CSL_UTOPIA2_TX_CLK_FAIL_INTR_ENABLE	Utopia2 transmit clock fail status
CSL_UTOPIA2_TX_QUEUE_STALL_INTR_ENABLE	Utopia2 transmit queue stall status
CSL_UTOPIA2_RX_CLK_PRES_INTR_ENABLE	Utopia2 receive clock present status
CSL_UTOPIA2_RX_CLK_FAIL_INTR_ENABLE	Utopia2 receive clock fail status
CSL_UTOPIA2_RX_QUEUE_STALL_INTR_ENABLE	Utopia2 receive queue stall status
CSL_UTOPIA2_TX_CLK_PRES_INTR_DISABLE	Utopia2 transmit clock present disable setup
CSL_UTOPIA2_TX_CLK_FAIL_INTR_DISABLE	Utopia2 transmit clock fail disable setup

CSL_UTOPIA2_TX_QUEUE_STALL_INTR_DISABLE	Utopia2 transmit queue stall disable setup
CSL_UTOPIA2_RX_CLK_PRES_INTR_DISABLE	Utopia2 receive clock present disable setup
CSL_UTOPIA2_RX_CLK_FAIL_INTR_DISABLE	Utopia2 receive clock fail disable setup
CSL_UTOPIA2_RX_QUEUE_STALL_INTR_DISABLE	Utopia2 receive queue stall disable setup

23.4.7 CSL_Utopia2Mphy

enum CSL_Utopia2Mphy
Utopia2 Multi - PHY Control Enumeration

Enumeration values

CSL_UTOPIA2_MPHY_DISABLE	Utopia2 Multi - PHY Disable
CSL_UTOPIA2_MPHY_ENABLE	Utopia2 Multi - PHY Enable

23.4.8 CSL_Utopia2PollingMode

enum CSL_Utopia2PollingMode
Utopia2 polling Mode Select Enumeration.

Enumeration values

CSL_UTOPIA2_ROUNDROBIN_POLLING	Utopia2 RoundRobin Polling Mode Select
CSL_UTOPIA2_FIXED_POLLING	Utopia2 Fixed Polling Mode Select

23.4.9 CSL_Utopia2SetupMode

enum CSL_Utopia2SetupMode
Utopia2 clock mode setup enumeration.

Enumeration values

CSL_UTOPIA2_REC_CLOCK	Utopia2 Receive clock mode selection
CSL_UTOPIA2_TX_CLOCK	Utopia2 Transmit clock mode selection

23.4.10 CSL_Utopia2SlidSlendMode

enum CSL_Utopia2SlidSlendMode
Utopia2 slave Id select enumeration.

Enumeration values

CSL_UTOPIA2_SLID_SLEND	Utopia2 slave Id

23.4.11 CSL_Utopia2UrenStatus

enum CSL_Utopia2UrenStatus
Utopia2 Receiver Control enumeration.

Enumeration values

CSL_UTOPIA2_UREN_DISABLE	Utopia2 Receiver Disable
CSL_UTOPIA2_UREN_ENABLE	Utopia2 Receiver Enable

23.4.12 CSL_Utopia2UxenStatus

enum CSL_Utopia2UxenStatus
Utopia2 transmitter Control enumeration.

Enumeration values

CSL_UTOPIA2_UXEN_DISABLE	Utopia2 Transmitter Disable
CSL_UTOPIA2_UXEN_ENABLE	Utopia2 Transmitter Enable

23.4.13 CSL_Utopia2ClkErrSts

enum CSL_Utopia2ClkErrSts
Utopia2 Clock error status enumeration.

Enumeration values

CSL_UTOPIA2_CTRL_XCFE	Command for Utopia2 XCFE
CSL_UTOPIA2_CTRL_XQSE	Command for Utopia2 XQSE
CSL_UTOPIA2_CTRL_XCPE	Command for Utopia2 XCPE
CSL_UTOPIA2_CTRL_RCFE	Command for Utopia2 RCFE
CSL_UTOPIA2_CTRL_RQSE	Command for Utopia2 RQSE
CSL_UTOPIA2_CTRL_RCPE	Command for Utopia2 RCPE

23.4.14 CSL_Utopia2ClkPendSts

enum CSL_Utopia2ClkPendSts
Utopia2 Clock error status enumeration.

Enumeration values

CSL_UTOPIA2_CTRL_ XCPP	Command for Utopia2 XCPP
CSL_UTOPIA2_CTRL_ XCFP	Command for Utopia2 XCFP
CSL_UTOPIA2_CTRL_ RCPP	Command for Utopia2 RCPP
CSL_UTOPIA2_CTRL_ RCFP	Command for Utopia2 RCFP

23.4.15 CSL_Utopia2IntrErrSts

enum CSL_Utopia2IntrErrSts
Utopia2 Clock error status enumeration.

Enumeration values

CSL_UTOPIA2_CTRL_XQIE0	Command for Utopia2 XQIE0
CSL_UTOPIA2_CTRL_XQIE1	Command for Utopia2 XQIE1
CSL_UTOPIA2_CTRL_XQIE2	Command for Utopia2 XQIE2
CSL_UTOPIA2_CTRL_XQIE3	Command for Utopia2 XQIE3
CSL_UTOPIA2_CTRL_XQIE4	Command for Utopia2 XQIE4
CSL_UTOPIA2_CTRL_XQIE5	Command for Utopia2 XQIE5
CSL_UTOPIA2_CTRL_XQIE6	Command for Utopia2 XQIE6
CSL_UTOPIA2_CTRL_XQIE7	Command for Utopia2 XQIE7
CSL_UTOPIA2_CTRL_ RQIE0	Command for Utopia2 RQIE0

CSL_UTOPIA2_CTRL_ RQIE1	Command for Utopia2 RQIE1
CSL_UTOPIA2_CTRL_ RQIE2	Command for Utopia2 RQIE2
CSL_UTOPIA2_CTRL_ RQIE3	Command for Utopia2 RQIE3
CSL_UTOPIA2_CTRL_ RQIE4	Command for Utopia2 RQIE4
CSL_UTOPIA2_CTRL_ RQIE5	Command for Utopia2 RQIE5
CSL_UTOPIA2_CTRL_ RQIE6	Command for Utopia2 RQIE6
CSL_UTOPIA2_CTRL_ RQIE7	Command for Utopia2 RQIE7

23.4.16 CSL_Utopia2IntrPendSts

enum CSL_Utopia2IntrPendSts
Utopia2 Clock error status enumeration.

Enumeration values

CSL_UTOPIA2_CTRL_XQIP0	Command for Utopia2 XQIP0
CSL_UTOPIA2_CTRL_XQIP1	Command for Utopia2 XQIP1
CSL_UTOPIA2_CTRL_XQIP2	Command for Utopia2 XQIP2
CSL_UTOPIA2_CTRL_XQIP3	Command for Utopia2 XQIP3
CSL_UTOPIA2_CTRL_XQIP4	Command for Utopia2 XQIP4
CSL_UTOPIA2_CTRL_XQIP5	Command for Utopia2 XQIP5
CSL_UTOPIA2_CTRL_XQIP6	Command for Utopia2 XQIP6
CSL_UTOPIA2_CTRL_XQIP7	Command for Utopia2 XQIP7
CSL_UTOPIA2_CTRL_ RQIP0	Command for Utopia2 RQIP0
CSL_UTOPIA2_CTRL_ RQIP1	Command for Utopia2 RQIP1
CSL_UTOPIA2_CTRL_ RQIP2	Command for Utopia2 RQIP2
CSL_UTOPIA2_CTRL_ RQIP3	Command for Utopia2 RQIP3
CSL_UTOPIA2_CTRL_ RQIP4	Command for Utopia2 RQIP4
CSL_UTOPIA2_CTRL_ RQIP5	Command for Utopia2 RQIP5
CSL_UTOPIA2_CTRL_ RQIP6	Command for Utopia2 RQIP6
CSL_UTOPIA2_CTRL_ RQIP7	Command for Utopia2 RQIP7

23.4.17 CSL_Utopia2RecUdc

enum CSL_Utopia2RecUdc
Utopia2 receive user defined cell enumeration.

Enumeration values

CSL_UTOPIA2_RUDC_EXTRA0	Utopia2 receive user defined cell extra bits 0
CSL_UTOPIA2_RUDC_EXTRA1	Utopia2 receive user defined cell extra bits 1
CSL_UTOPIA2_RUDC_EXTRA2	Utopia2 receive user defined cell extra bits 2
CSL_UTOPIA2_RUDC_EXTRA3	Utopia2 receive user defined cell extra bits 3
CSL_UTOPIA2_RUDC_EXTRA4	Utopia2 receive user defined cell extra bits 4
CSL_UTOPIA2_RUDC_EXTRA5	Utopia2 receive user defined cell extra bits 5
CSL_UTOPIA2_RUDC_EXTRA6	Utopia2 receive user defined cell extra bits 6
CSL_UTOPIA2_RUDC_EXTRA7	Utopia2 receive user defined cell extra bits 7
CSL_UTOPIA2_RUDC_EXTRA8	Utopia2 receive user defined cell extra bits 8
CSL_UTOPIA2_RUDC_EXTRA9	Utopia2 receive user defined cell extra bits 9
CSL_UTOPIA2_RUDC_EXTRA10	Utopia2 receive user defined cell extra bits 10

CSL_UTOPIA2_RUDC_EXTRA11	Utopia2 receive user defined cell extra bits 11

23.4.18 CSL_ Utopia2TxUdc

enum CSL_ Utopia2TxUdc
Utopia2 transmit user defined cell enumeration.

Enumeration values

CSL_UTOPIA2_XUDC_EXTRA0	Utopia2 transmit user defined cell extra bits 0
CSL_UTOPIA2_XUDC_EXTRA1	Utopia2 transmit user defined cell extra bits 1
CSL_UTOPIA2_XUDC_EXTRA2	Utopia2 transmit user defined cell extra bits 2
CSL_UTOPIA2_XUDC_EXTRA3	Utopia2 transmit user defined cell extra bits 3
CSL_UTOPIA2_XUDC_EXTRA4	Utopia2 transmit user defined cell extra bits 4
CSL_UTOPIA2_XUDC_EXTRA5	Utopia2 transmit user defined cell extra bits 5
CSL_UTOPIA2_XUDC_EXTRA6	Utopia2 transmit user defined cell extra bits 6
CSL_UTOPIA2_XUDC_EXTRA7	Utopia2 transmit user defined cell extra bits 7
CSL_UTOPIA2_XUDC_EXTRA8	Utopia2 transmit user defined cell extra bits 8
CSL_UTOPIA2_XUDC_EXTRA9	Utopia2 transmit user defined cell extra bits 9
CSL_UTOPIA2_XUDC_EXTRA10	Utopia2 transmit user defined cell extra bits 10
CSL_UTOPIA2_XUDC_EXTRA11	Utopia2 transmit user defined cell extra bits 11

23.5 Macros

#define CSL_UTOPIA2_REC_CMD (2)
Command for Utopia2 Receiver

#define CSL_UTOPIA2_XMT_CMD (1)
Command for Utopia2 Transmitter

23.6 Typedefs

typedef CSL_Utopia2ClkSetup CSL_Utopia2ClkSetup
Utopia2 Clock Setup Command Structure.

typedef CSL_Utopia2ErrorSetup CSL_Utopia2ErrorSetup

typedef CSL_Utopia2UcrSetup CSL_Utopia2UcrSetup
Utopia2 'Utopia2 Control Register' Setup Command Structure.

typedef CSL_Utopia2HwSetup CSL_Utopia2HwSetup
Utopia2 Hardware Setup structure.

typedef CSL_Utopia2GlobalGetSetup CSL_Utopia2GlobalGetSetup
Utopia2 Hardware Setup Structure.

typedef CSL_Utopia2Obj CSL_Utopia2Obj
This structure/object holds the context of the instance of UTOPIA2 opened using CSL_utopia2Open() function. Pointer to this object is passed as UTOPIA2 Handle to all UTOPIA2 CSL APIs. CSL_utopia2Open() function initializes this structure based on the parameters passed.

typedef CSL_Utopia2Obj * CSL_Utopia2Handle

Chapter 24
BWMNGMT Module

Topics

24.1 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

24.2 Functions

This section lists the functions available in the BWMNGMT module.

24.2.1 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_bwmngmtInit (NULL);
```

24.2.2 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 Invalid parameters

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;

hBwmngmt = CSL_bwmngmtOpen (&bwmngmtObj,
                            CSL_BWMNGMT,
                            NULL,
                            &status
                            );
```

24.2.3 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 that order before *CSL_bwmngmtClose()* can be called.

Post Condition
None

Modifies
CSL_BwmngmtObj structure instance values

Example
```
CSL_Status            status;
CSL_BwmngmtObj        bwmngmtObj;
CSL_BwmngmtHandle     hBwmngmt;

hBwmngmt = CSL_bwmngmtOpen (&bwmngmtObj,CSL_BWMNGMT,NULL,&status);

...

CSL_bwmngmtClose (hBwmngmt);
```

24.2.4 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 that order before this function can be called. 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

The **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;
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);
```

24.2.5 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 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 that order before this function can be called.

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

Modifies
Modifies the second parameter.

Example

```
CSL_BwmngmtHandle     hBwmngmt;
```

```
CSL_BwmngmtHwSetup    hwSetup;
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
...
CSL_bwmngmtGetHwSetup(hBwmngmt, &hwSetup);
```

24.2.6 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
None

Modifies
None

Example

```
CSL_BwmngmtHandle         hBwmngmt;
CSL_BwmngmtHwControlCmd   cmd;
```

```
...
status = CSL_bwmngmtHwControl (hBwmngmt, cmd, NULL);
...
```

24.2.7 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 that order before this function can be called

Post Condition
None

Modifies
None

Example

```
CSL_BwmngmtHandle          hBwmngmt;
CSL_BwmngmtHwStatusQuery   query;
void                       response;

...
status = CSL_bwmngmtGetHwStatus(hBwmngmt, query, &response);
...
```

24.2.8 CSL_bwmngmtGetBaseAddress

CSL_Status CSL_bwmngmtGetBaseAddress(CSL_InstNum *bwmngmtNum*,
CSL_BwmngmtParam * *pBwmngmtParam*,
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. MMR's go to an alternate location.

Arguments

```
bwmngmtNum    Specifies the instance of the bwmngmt to be opened.

pBwmngmtParam 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 bwmngmt.
- CSL_ESYS_FAIL - The instance number is invalid.
- CSL_ESYS_INVPARAMS - Inavlid 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_BwmngmtBaseAddress  baseAddress;

     ...
status = CSL_bwmngmtGetBaseAddress(CSL_BWMNGMT, NULL,
&baseAddress);
```

24.3 Data Structures

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

24.3.1 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.

24.3.2 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

24.3.3 CSL_BwmngmtBaseAddress

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

Field Documentation

CSL_BwmngmtRegsOvly CSL_BwmngmtBaseAddress::regs
Base-address of the Configuration registers of BWMNGMT

24.3.4 CSL_BwmngmtParam

Detailed Description
BWMNGMT specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_BwmngmtParam:: flags
Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

24.3.5 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. The below declaration is just a place-holder for future implementation.

24.4 Enumerations

This section lists the enumerations available in the BWMNGMT module.

24.4.1 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

24.4.2 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

24.4.3 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

24.4.4 CSL_BwmngmtHwStatusQuery

enum CSL_BwmngmtHwStatusQuery
Enumeration for Hardware status query

Enumeration values:

PLACEHOLDER0	Placeholder for future implementation

24.4.5 CSL_BwmngmtHwControlCmd

enum CSL_BwmngmtHwControlCmd
Enumeration for Hardware control command
Enumeration values:

PLACEHOLDER2	Placeholder for future implementation

24.5 Macros

#define CSL_BWMNGMT_HWSETUP_DEFAULTS
Value:

```
{  \
        (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 #define CSL_BWMNGMT_HWSETUP_DEFAULTS is meant to simplify the implementation in C code by the customer

24.6 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 25
CACHE Module

Topics

25.1 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.

25.2 Functions

This section lists the functions available in the CACHE module.

25.2.1 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);
```

25.2.2 CACHE_disableCaching

void CACHE_enableCaching (CE_MAR *mar*)

Description
This function disables caching for a specific memory region.

Arguments

mar	Memory region for which cache is to be disabled.

Return Value
None

Pre Condition
None

Post Condition

Memory region is now *not* cacheable.

Modifies
MAR registers

Example

```
CACHE_disableCaching (CACHE_EMIFB_CE15);
```

25.2.3 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();
...
```

25.2.4 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();
...
```

25.2.5 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Freeze L1 cache

Modifies
L1DCC and L1PCC registers

Example

```
...
CACHE_L1_Freeze oldFreezeState ;

oldFreezeState = CACHE_freezeL1();
```

25.2.6 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Unfreeze the L1 cache

Modifies
L1DCC and L1PCC registers

Example

```
...
CACHE_L1_Freeze oldFreezeState;

oldFreezeState = CACHE_unfreezeL1();
...
```

25.2.7 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);
...
```

25.2.8 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Freeze L1P cache

Modifies
L1PCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;

oldFreezeState = CACHE_freezeL1p();
...
```

25.2.9 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 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_L1P_FREEZE` - Old Freeze State of L1P Cache
- `CACHE_L1P_NORMAL` - Normal State of L1P Cache

Pre Condition
The CACHE must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Unreeze L1P cache

Modifies
L1PCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;

oldFreezeState = CACHE_unfreezeL1p();
...
```

25.2.10 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
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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate L1P cache

Modifies
L1PIBAR and L1PIWC registers

Example

```
...
CACHE_invL1p ((Uint32*)(0x1000), 200, CACHE_NOWAIT);
...
```

25.2.11 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate all L1P cache

Modifies
L1PINV register

Example

```
...
CACHE_invAllL1p (CACHE_NOWAIT);
...
```

25.2.12 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);
...
```

25.2.13 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Freeze L1D cache

Modifies
L1DCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;

oldFreezeState = CACHE_freezeL1d();
...
```

25.2.14 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Unfreeze L1D cache

Modifies
L1DCC register

Example

```
...
CACHE_L1_Freeze oldFreezeState;

oldFreezeState = CACHE_unfreezeL1d();
...
```

25.2.15 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,
a. The start of the range that needs to be written back is programmed into L1DWBAR.
b. 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback L1D cache

Modifies
L1DWWC and L1DWBAR registers

Example

```
...
CACHE_wbL1d((Uint32*)(0x1000), 200, CACHE_NOWAIT);
```

25.2.16 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate the L1D cache

Modifies
L1DIWC and L1DIBAR registers

Example

```
...
CACHE_invL1d ((Uint32*)(0x1000), 200, CACHE_NOWAIT);
```

25.2.17 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback and invalidate the L1D cache

Modifies
L1DWIWC and L1DWIBAR registers

Example

```
...
CACHE_wbInvL1d ((Uint32*)(0x1000),200,CACHE_NOWAIT);
```

25.2.18 CACHE_wbAllL1d

void CACHE_wbAllL1d (CACHE_Wait *wait*)

Description
This function issues an L1D writeback all command to the cache controller. . If CACHE_wbAllL1d was also a previous cache operation which 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 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback all the L1D cache

Modifies
L1DWB register

Example

```
        ...
        CACHE_wbAllL1d (CACHE_NOWAIT);
        ...
```

25.2.19 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate the all L1D cache

Modifies
L1DINV register

Example

```
        ...
        CACHE_invAllL1d (CACHE_NOWAIT);
        ...
```

25.2.20 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback and invalidate all L1D cache

Modifies
L1DWBINV register

Example

```
...
CACHE_wbInvAllL1d (CACHE_NOWAIT);
...
```

25.2.21 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
Set the L2 cache size

Modifies
L2CFG register

Example

```
        ...
        CACHE_L2Size oldSize;

        oldSize = CACHE_setL2Size(CACHE_L2_32KCACHE);
        ...
```

25.2.22 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Set L2 cache mode

Modifies
L2CFG register

Example

```
...
CACHE_L2Mode oldMode;

oldMode = CACHE_setL2Mode(CACHE_L2_NORMAL);
...
```

25.2.23 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback the L2 cache

Modifies
L2WWC and L2WBAR registers

Example

```
...
CACHE_wbL2((Uint32*)(0x1000), 200, CACHE_NOWAIT);
```

25.2.24 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 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 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate the L2 cache

Modifies
L2IBAR and L2IWC registers

Example

```
...
CACHE_invL2((Uint32*)(0x1000), 200, CACHE_NOWAIT);
...
```

25.2.25 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback and invalidate the L2 cache

Modifies
L2WIBAR and L2WIWC registers

Example

```
...
CACHE_wbInvL2((Uint32*)(0x1000), 200, CACHE_NOWAIT);
```

25.2.26 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback all L2 cache

Modifies
L2WB register

Example

```
...
CACHE_wbAllL2(CACHE_NOWAIT);
...
```

25.2.27 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Invalidate all L2 cache

Modifies
L2INV register

Example

```
...
CACHE_invAllL2(CACHE_NOWAIT);
...
```

25.2.28 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 must be successfully enabled via CACHE_enableCaching() before calling this function

Post Condition
Writeback and invalidate all the L2 cache

Modifies
L2WBINV register

Example

```
...
CACHE_wbInvAllL2(CACHE_NOWAIT);
...
```

25.3 Enumerations

This section lists the enumerations available in the CACHE module.

25.3.1 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_CE00	EMIF ranges from 0xE0000000 – 0xE0FFFFFF
CACHE_EMIFA_CE01	EMIF ranges from 0xE1000000 – 0xE1FFFFFF
CACHE_EMIFA_CE02	EMIF ranges from 0xE2000000 – 0xE2FFFFFF
CACHE_EMIFA_CE03	EMIF ranges from 0xE3000000 – 0xE3FFFFFF
CACHE_EMIFA_CE04	EMIF ranges from 0xE4000000 – 0xE4FFFFFF
CACHE_EMIFA_CE05	EMIF ranges from 0xE5000000 – 0xE5FFFFFF
CACHE_EMIFA_CE06	EMIF ranges from 0xE6000000 – 0xE6FFFFFF
CACHE_EMIFA_CE07	EMIF ranges from 0xE7000000 – 0xE7FFFFFF
CACHE_EMIFA_CE08	EMIF ranges from 0xE8000000 – 0xE8FFFFFF
CACHE_EMIFA_CE09	EMIF ranges from 0xE9000000 – 0xE9FFFFFF
CACHE_EMIFA_CE10	EMIF ranges from 0xEA000000 – 0xEAFFFFFF
CACHE_EMIFA_CE11	EMIF ranges from 0xEB000000 – 0xEBFFFFFF
CACHE_EMIFA_CE12	EMIF ranges from 0xEC000000 – 0xECFFFFFF
CACHE_EMIFA_CE13	EMIF ranges from 0xED000000 – 0xEDFFFFFF
CACHE_EMIFA_CE14	EMIF ranges from 0xEE000000 – 0xEEFFFFFF
CACHE_EMIFA_CE15	EMIF ranges from 0xEF000000 – 0xEFFFFFFF
CACHE_EMIFB_CE00	EMIF ranges from 0xF0000000 – 0xF0FFFFFF
CACHE_EMIFB_CE01	EMIF ranges from 0xF1000000 – 0xF1FFFFFF
CACHE_EMIFB_CE02	EMIF ranges from 0xF2000000 – 0xF2FFFFFF
CACHE_EMIFB_CE03	EMIF ranges from 0xF3000000 – 0xF3FFFFFF
CACHE_EMIFB_CE04	EMIF ranges from 0xF4000000 – 0xF4FFFFFF
CACHE_EMIFB_CE05	EMIF ranges from 0xF5000000 – 0xF5FFFFFF
CACHE_EMIFB_CE06	EMIF ranges from 0xF6000000 – 0xF6FFFFFF
CACHE_EMIFB_CE07	EMIF ranges from 0xF7000000 – 0xF7FFFFFF
CACHE_EMIFB_CE08	EMIF ranges from 0xF8000000 – 0xF8FFFFFF
CACHE_EMIFB_CE09	EMIF ranges from 0xF9000000 – 0xF9FFFFFF
CACHE_EMIFB_CE10	EMIF ranges from 0xFA000000 – 0xFAFFFFFF
CACHE_EMIFB_CE11	EMIF ranges from 0xFB000000 – 0xFBFFFFFF
CACHE_EMIFB_CE12	EMIF ranges from 0xFC000000 – 0xFCFFFFFF
CACHE_EMIFB_CE13	EMIF ranges from 0xFD000000 – 0xFDFFFFFF
CACHE_EMIFB_CE14	EMIF ranges from 0xFE000000 – 0xFEFFFFFF
CACHE_EMIFB_CE15	EMIF ranges from 0xFF000000 – 0xFFFFFFFF

25.3.2 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)

25.3.3 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

25.3.4 CACHE_L1Size

enum CACHE_L1Size
Enumeration for L1 (P or D) 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

25.3.5 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

25.3.6 CACHE_L2Mode

enum CACHE_L2Mode

Enumeration for L2 Modes.

Enumeration values:

CACHE_L2_NORMAL	Enabled/Normal Mode
CACHE_L2_FREEZE	Freeze Mode

25.4 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)

Value:

```
( ( ( ( (ELCNT) * (ELSIZE)\
        + CACHE_##CACHE##_LINESIZE - 1\
      ) / CACHE_##CACHE##_LINESIZE\
      * CACHE_##CACHE##_LINESIZE\
    ) + (ELSIZE) - 1\
  ) / (ELSIZE)\
)
```

Cache Round to Line size

Chapter 26
CFG Module

Topics

26.1 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.

26.2 Functions

This section lists the functions available in the CFG module.

26.2.1 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);
```

26.2.2 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);
```

26.2.3 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 that order before *CSL_cfgClose()* can be called.

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_Status        status;
CSL_CfgObj        cfgObj;
CSL_CfgHandle     hCfg;

hCfg = CSL_cfgOpen (&cfgObj, CSL_MEMPROT_CONFIG, NULL, &status);

CSL_cfgClose(hCfg);
```

26.2.4 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 that order before *CSL_cfgHwControl()* can be called.

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;
CSL_CfgObj            cfgObj;
CSL_CfgHwControlCmd   cmd = CSL_CFG_CMD_CLEAR;

hCfg = CSL_cfgOpen (&cfgObj, CSL_MEMPROT_CONFIG, NULL, &status);

status = CSL_cfgHwControl(hCfg, cmd, NULL);
```

26.2.5 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 that order before *CSL_cfgGetHwStatus()* can be called.

Post Condition
None

Modifies

Third parameter “response” vlaue

Example

```
CSL_CfgHandle          hCfg;
CSL_CfgHwStatusQuery   query = CSL_CFG_QUERY_FAULT_ADDR;
CSL_Status             status;
CSL_CfgObj             cfgObj;
```

```
    void                    response;

    hCfg = CSL_cfgOpen (&cfgObj, CSL_MEMPROT_CONFIG, NULL, &status);

    status = CSL_cfgGetHwStatus(hCfg, query, &response);
```

26.2.6 CSL_cfgGetBaseAddress

CSL_Status CSL_cfgGetBaseAddress (CSL_InstNum cfg*Num*,
CSL_CfgParam * *p*Cfg*Param*,
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);
```

26.3 Data Structures

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

26.3.1 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

26.3.2 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

26.3.3 CSL_CfgBaseAddress

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

Field Documentation

CSL_CfgRegsOvly CSL_CfgBaseAddress:: regs
Base-address of the Configuration registers of CFG

26.3.4 CSL_CfgParam

Detailed Description
CFG specific parameters. Present implementation doesn't have any specific parameters.

Field Documentation

CSL_BitMask16 CSL_CfgParam:: flags
Bit mask to be used for module specific parameters. The below declaration is just a place-holder for future implementation.

26.3.5 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. The below declaration is just a place-holder for future implementation

26.4 Enumerations

This section lists the enumerations available in the CFG module.

26.4.1 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

26.4.2 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:
*(Uint16 *)*

CSL_CFG_QUERY_FAULT_STATUS Status query command to get the Status information of CSL_CfgStatus.
Parameters:
*(CSL_CfgStatus *)*

26.5 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

26.6 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 27
CHIP Module

Topics

27.1 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.

27.2 Functions

This section lists the functions available in the CHIP module.

27.2.1 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 oldamr;
oldamr = CSL_chipWriteReg (CSL_CHIP_AMR, 56);
```

27.2.2 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 amrVal;
amrVal = CSL_chipReadReg (CSL_CHIP_AMR, 56);
```

27.3 Enumerations

This section lists the enumerations available in the CHIP module.

27.3.1 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 28
IDMA MODULE

Topics

28.1 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.

28.2 Functions

This section lists the functions available in the IDMA module.

28.2.1 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);
```

28.2.2 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 IDMA_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];
...
```

```
void main (void)
{
...
// Copy src to dst1 - 80 bytes - 20 words
IDMA1_copy(src, dst1, 80);
... }
```

28.2.3 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 IDMA_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];

void main (void)
{
...

IDMA1_fill(dst1, 80, 0xAAAABABA);
...
}
```

28.2.4 IDMA1_getStatus

Uint32 IDMA1_getStatus (void)

Description
IDMA1_getStatus() gets the active and pending status of IDMA 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();
```

28.2.5 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 that order before this API can be invoked.

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];

void main (void)
{
Uint32        stat;


...
IDMA1_fill (dst, 80, 0xAAAAAAAA);
stat = IDMA1_getStatus();
IDMA1_wait();
}
```

28.2.6 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);
```

28.2.7 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);
```

28.2.8 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);
```

28.2.9 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();
```

28.2.10 IDMA0_configArgs

void IDMA0_configArgs (Uint32 *mask*,
Uint32 * *src*,
Uint32 * *dst*,
Uint32 *count*

)

Description
IDMA0_configArgs() - Configures IDMA 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
cnt	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();
```

28.2.11 IDMA0_getStatus

Uint32 IDMA0_getStatus (void)

Description
IDMA0_getStatus() gets the active and pending status of IDMA 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();
```

28.2.12 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 that order before this API can be invoked.

Post Condition
Completion of previous transfer

Modifies
IDMA channel 0 registers

Example

```
Uint32        stat;
...

stat = IDMA0_getStatus();
IDMA0_wait();
```

28.2.13 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);
```

28.3 Data Structures

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

28.3.1 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

28.3.2 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

28.4 Enumerations

This section lists the enumerations available in the IDMA module.

28.4.1 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

28.4.2 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

28.4.3 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

Chapter 29
MEMPROT Module

Topics

29.1 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.

.

29.2 Functions

This section lists the functions available in the MEMPROT module.

29.2.1 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
This function should be called before using any of the CSL APIs in the Memory Protection module.
Note: As Memory Protection doesn't have any context based information,
the function just returns CSL_SOK. User is expected to pass NULL.

Post Condition
None

Modifies
None

Example

```
CSL_memprotInit (NULL);
```

29.2.2 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);
```

29.2.3 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
CSL_memprotInit(), CSL_memprotOpen() must be opened prior to this call.

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;
CSL_MemprotObj      mpL2Obj;

hMemprot = CSL_memprotOpen(&mpL2Obj,
                           CSL_MEMPROT_L2,
                           NULL,
                           &status);

status = CSL_memprotClose(hMemprot);
```

29.2.4 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 that order before this function can be called. 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
1. MEMPROT object structure is populated
2. The status is returned in the status variable. If status returned is

- CSL_SOK - Valid MEMPROT handle is returned
- CSL_ESYS_INVPARAMS - Invalid parameter
- CSL_ESYS_BADHANDLE - Invalid Handle

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);
```

29.2.5 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()* and*CSL_memprotOpen()* must be called successfully in that order before *CSL_memprotGetHwSetup()* can be called. 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
The hardware set up structure will be populated with values from the registers.

Modifies
Second parameter setup

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);
    CSL_memprotGetHwSetup(hmpL2,&L2MpGetSetup);
```

29.2.6 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 that order before *CSL_memprotHwControl()* can be called. 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,L2MpGetSetup;
CSL_MemprotLockStatus   lockStat;
CSL_MemprotPageAttr     pageAttr;
CSL_MemprotFaultStatus  queryFaultStatus;

// 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)
{
 CSL_memprotHwControl(hmpL2,CSL_MEMPROT_CMD_UNLOCK,key);
}
```

29.2.7 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_INVPARAMS - Invalid parameter
- CSL_ESYS_INVQUERY - Invalid Query

Pre Condition

Both *CSL_memprotInit()* and *CSL_memprotOpen()* must be called successfully in that order before *CSL_memprotGetHwStatus()* can be called. 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,L2MpGetSetup;
CSL_MemprotLockStatus lockStat;
CSL_MemprotPageAttr pageAttr;
CSL_MemprotFaultStatus queryFaultStatus;

// 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);
```

29.2.8 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 - Successfully retrieved base address
- 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);
```

29.3 Data Structures

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

29.3.1 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.

29.3.2 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.

29.3.3 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

29.3.4 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

29.3.5 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

29.3.6 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

29.3.7 CSL_MemprotParam

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

29.4 Enumerations

This section lists the enumerations available in the MEMPROT module.

29.4.1 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 *)*

29.4.2 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*)*

29.4.3 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

29.5 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

Chapter 30
PWRDWN Module

Topics

30.1 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.

30.2 Functions

This section lists the functions available in the PWRDWN module.

30.2.1 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

```
CSL_pwrdwnInit(NULL);
```

30.2.2 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. There are three instance of the PWRDWN available. So, the value for this parameter will be based on the instance.
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 parameters.

2. Pwrdwn object structure is populated.

Modifies

1. The status variable
2. pwrdwn object structure

Example

```
CSL_PwrdwnObj    pwrObj;
CSL_Status       status;
CSL_PwrdwnConfig pwrConfig;
CSL_PwrdwnHandle hPwr;

// Init Module
CSL_pwrdwnInit(NULL);

// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen (&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Config structure
...
```

30.2.3 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

1. 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_Status       status;
CSL_PwrdwnConfig pwrConfig;
CSL_PwrdwnHandle hPwr;

// Init Module
CSL_pwrdwnInit(NULL);

// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen(&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Config structure
...
// Close
CSL_pwrdwnClose(hPwr);
```

30.2.4 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 called 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_Status        status;
CSL_PwrdwnHwSetup pwrSetup;
CSL_PwrdwnHandle  hPwr;

// Init Module
CSL_pwrdwnInit(NULL);

// 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);

// Close handle
CSL_pwrdwnClose(hPwr);
```

30.2.5 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 - Invalid Parameters.

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_Status        status;
       CSL_PwrdwnHwSetup pwrSetup, querySetup;
       CSL_PwrdwnHandle  hPwr;

       // Init Module
   CSL_pwrdwnInit(NULL);

       // 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
       CSL_pwrdwnGetHwSetup(hPwr,&querySetup);

       // Close handle
       CSL_pwrdwnClose(hPwr);
```

30.2.6 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_Status              status;
CSL_PwrdwnHwSetup       pwrSetup;
CSL_PwrdwnHandle        hPwr;
CSL_PwrdwnPortData      pageSleep;

pageSleep.portNum = 0x0;

// Init Module
CSL_pwrdwnInit(NULL);

// 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
CSL_pwrdwnGetHwStatus(  hPwr,
                        CSL_PWRDWN_QUERY_PAGE0_STATUS,
                        &pageSleep);
// Close handle
CSL_pwrdwnClose(hPwr);
```

30.2.7 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
CSL_pwrdwnInit(NULL);

// Opening a handle for the Module
hPwr = CSL_pwrdwnOpen (&pwrObj, CSL_PWRDWN, NULL, &status);

// Setup the arguments for the Config structure
...
// Setup
CSL_pwrdwnHwSetupRaw(hPwr,&pwrConfig);
// Close handle

CSL_pwrdwnClose(hPwr);
```

30.2.8 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 - Successfully retrieved base address
- 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_PwrdwnHandle       hPwrdwn;
CSL_PwrdwnBaseAddress  baseAddress;
CSL_PwrdwnParam        params;

CSL_pwrdwnGetBaseAddress(CSL_PWRDWN, &params, &baseAddress) ;
```

30.2.9 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
CSL_pwrdwnInit(NULL);

// 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;

CSL_pwrdwnHwControl(hPwr,CSL_PWRDWN_CMD_PAGE0_SLEEP,&pageSleep);
// Close handle
CSL_pwrdwnClose(hPwr);
```

30.3 Data Structures

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

30.3.1 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

30.3.2 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

30.3.3 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.

30.3.4 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

30.3.5 CSL_PwrdwnParam

Detailed Description
Module specific parameters. None in this implementation.

Field Documentation

void* CSL_PwrdwnParam::futureUse
Perhaps useful for future use

30.3.6 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

30.3.7 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

30.3.8 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

30.4 Enumerations

This section lists the enumerations available in the PWRDWN module.

30.4.1 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* *)

30.4.2 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* *)

30.5 Typedefs

typedef CSL_PwrdwnObj *CSL_PwrdwnHandle
This is a pointer to CSL_PwrdwnObj & is passed as the first parameter to all PWRDWN CSL APIs

Chapter 31
TSC Module

Topics

31.1 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.

31.2 Functions

This section lists the functions available in the TSC module.

31.2.1 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 begin incrementing once per CPU cycle. You cannot reset the time stamp counter.

Arguments
None

Return Value
None

Pre Condition
None

Post Condition
Time Stamp Counter value starts incrementing

Modifies
None

Example

```
        CSL_tscEnable();
```

31.2.2 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
None

Post Condition
None

Modifies
None

Example

```
CSL_Uint64      counterVal;

...
CSL_tscEnable();
counterVal  = CSL_tscRead ();
```