TEXAS INSTRUMENTS

Application Report

Using the Chip Support Register Configuration Macros for DM648

Platform Support Group

ABSTRACT

This document describes the Chip Support Register Configuration files provided for some Digital Media Processors (DMPs) and C64x+ Digital Signal Processors (DSPs). This layer provides low-level register and bit field descriptions for the device and its peripherals, and a set of macros for basic register configuration. It may be used as a foundation for building complex drivers or on its own to perform register configuration and check peripheral status.

Contents

1 Overview of the Chip Support Register Configuration Layer 2
  1.1 Chip Support Register Configuration File Structure 2
  1.2 Common File Attributes 4
  1.3 Peripheral-Specific File Attributes 4
    1.3.1 Register Overlay Structure 4
    1.3.2 Field Definitions 5
    1.3.3 Bit Field Definition Example 6
2 Macro Reference 7
3 Examples 10
  3.1 DDR2 Example 10
  3.2 PLLC Example 13
  3.3 GPIO Example 15
4 References 17

1 Overview of the Chip Support Register Configuration Layer

The Chip Support Register Configuration files provide register configuration support for each of the peripheral modules on selected Digital Media Processor (DMPs) and C64x+ DSPs through a set of C header files delivered in the Platform Support Package (PSP). Module-specific files provide register and bit field descriptions for a given peripheral, and a common file provides macros to read and modify hardware registers. Other common and system files provide for other device-specific definitions. See Table 1 for a list of supported devices.

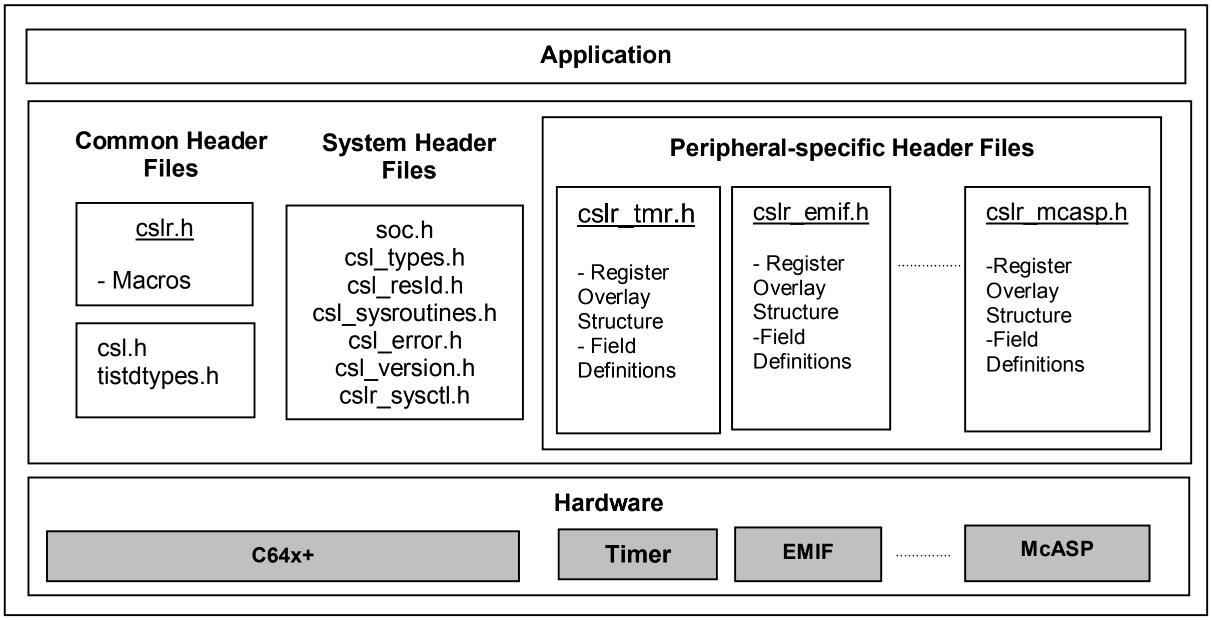
Family	Devices	Delivery Mechanism
TMS320DM648 DMP	DM648	DSP/BIOS PSP for the DM648 Digital Video Development Platform (DVDP)
TMS320C645x DSP	C6452	DSP/BIOS PSP for C6452 EVM

Table 1. Chip Support Register Configuration Layer Supported Devices

1.1 Chip Support Register Configuration File Structure

The Chip Support Register Configuration files are made up of three types of header files: common files, system files, and peripheral-specific files. These files are summarized in Table 2 and in the figure below.

The Chip Support Register Configuration files are delivered in a Platform Support Package (PSP), in the directory */soc/*<device>*/dsp/inc*, where *<device>* is the member of the family with EVM support. For example, the DM648 devices shown in Table 1 are supported by the header files in the directory */soc/dm648/dsp/* in the DM648 PSP.



Chip Support Register Configuration File Structure

Common files are independent of a device or family, and independent of any specific registers. These define standard data types or macros. System files are specific to the device. They define peripheral instances, version information, error types, interrupt event IDs, interrupt routines, DMA channel structure, and they provide data types which may be specific to the device family.

In addition, each peripheral or module type is supported by a register configuration layer header file, which contains a register overlay structure and field definitions. The naming convention for peripheral-specific header files is cslr_<*per*>.h, where <*per*> is the abbreviation for the peripheral. For example, cslr_gpio.h is the header file for the GPIO peripheral.

> **Note:** Some peripherals are made up of multiple header file components. For example TMS320DM648 ethernet peripheral has multiple subcomponents and each of them would have CSLR file.

A system-level register layer header file named cslr_sysctl.h contains the register overlay structure and field definitions for the system module registers used for device configuration. This file also includes control registers for the timer, EDMA transfer controller and DDR2 memory controller. The other registers for these peripherals are supported in their respective peripheral header files. The memory map for the system module registers is summarized in the device datasheet.

The user need only include the peripheral header files and common header files required for the application.

File Name	File Type	Description
csl.h	Common	System initialization function.
cslr.h	Common	Macros for register and bit field manipulation.
tistdtypes.h	Common	Standard data types common to TI software products.
soc.h	Device	Peripheral instance definitions, peripheral base addresses, and other definitions common to the device, such as interrupt event IDs and DMA channel parameters.
csl_types.h	Device	Additional data types.
csl_resId.h	Device	Resource IDs and IO masks for peripheral instances.
csl_sysroutines.h	Device	Interrupt restore and disable routines.
csl_error.h	Device	Global and peripheral-specific error codes.
csl_version.h	Device	CSL version and device ID strings.
cslr_sysctl.h	Device	Register overlay structure and field definitions for system level features such as pin multiplexing, device boot, Switch Central Resource (SCR), JTAG and power down control. Also supports system level control registers for certain peripherals: EDMA transfer controller, timer, DDR2 memory controller.
cslr_<*per*>.h	Peripheral	Peripheral or module-specific header files, where <*per*> is the abbreviation for the peripheral.

Table 2. Chip Support Register Configuration Files

Key attributes of some of these files are described in more detail in the sections that follow.

1.2 Common File Attributes

The Chip Support Register Configuration layer defines eight macros in the file cslr.h. These macros allow the programmer to create, read, or write bit fields within a register. There are three different types of services: field make, field extract, and field insert. The macros summarized in Table 3 are described in detail in section 2.

Macro	Brief Description	Page
CSL_FMK	Field Make	7
CSL_FMKT	Field Make Token	7
CSL_FMKR	Field Make Raw	7
CSL_FEXT	Field Extract	8
CSL_FEXTR	Field Extract Raw	8
CSL_FINS	Field Insert	8
CSL_FINST	Field Insert Token	9
CSL_FINSR	Field Insert Raw	9

Table 3. Register Configuration Macros in cslr.h

Field make macros are used to create a register value from given input, and are written to the hardware register with the pointer to the register member in the Register Overlay Structure. Field make macros may be combined with OR operations in order to modify more than one field or the entire register. Unlike the field make macros, the field insert macros pass the register pointer as an argument, thus modify the specified register directly. Field extract macros read the register and return the value right-justified.

Raw macros provide the flexibility to modify or read one, multiple, or partial bit fields, because they designate the range of affected bits by location.

For macros that pass field name as an argument, the format for field name described in section 1.3.2 applies.

1.3 Peripheral-Specific File Attributes

1.3.1 Register Overlay Structure

The register overlay structure is defined for each peripheral in its register configuration layer header file, named cslr_<*per*>.h, where <*per*> is the abbreviation for the peripheral. The register overlay structure defines peripheral hardware registers, matching the hardware memory in sequence and register offset.

The naming convention of the register overlay structure type is CSL_<*Per*>Regs, where <*Per*> is the abbreviated peripheral type. The pointer type for the register overlay structure has the convention *CSL_<*Per*>RegsOvly. For example, the register overlay structure type for a Host Port Interface (HPI) is CSL_HpiRegs, and the pointer is *CSL_HpiRegsOvly.

By assigning the base address of the peripheral instance to the structure pointer, the structure members can be used to access the peripheral registers.

The format of the register overlay structure is as follows:

```
typedef struct  {
    volatile Uint32 REGISTER_1;
    volatile Uint32 REGISTER_2;
    :
    volatile Uint32 REGISTER_N;
} CSL_<Per>Regs;
```

The format of the register overlay structure pointer type definition is as follows:

```
typedef volatile CSL_<Per>Regs  *CSL_<Per>RegsOvly;
```

As an example, here are the register overlay structure and the pointer type definition for the DM648, from the file cslr_tmr.h:

```
/**************************************************************************\
* Register Overlay Structure
\**************************************************************************/
typedef struct  {
    volatile Uint32 PID12;
    volatile Uint32 EMUMGT_CLKSPD;
    volatile Uint8 RSVD0[8];
    volatile Uint32 TIM12;
    volatile Uint32 TIM34;
    volatile Uint32 PRD12;
    volatile Uint32 PRD34;
    volatile Uint32 TCR;
    volatile Uint32 TGCR;
    volatile Uint32 WDTCR;
} CSL_TmrRegs;

/**************************************************************************\
* Overlay structure typedef definition
\**************************************************************************/
typedef volatile CSL_TmrRegs         *CSL_TmrRegsOvly;
```

1.3.2 Field Definitions

The register configuration layer header file for each peripheral also contains definitions for field mask and shift values and hardware reset values for registers and bit fields.

The naming convention for these constants is:

CSL_<*PER*>_<*REG*>_<*FIELD*>_<*ACTION*>, where <*PER*> is the peripheral name, <*REG*> is the register name, <*FIELD*> is the name of the bit field. <*ACTION*> stands for MASK, SHIFT, RESETVAL, or a constant token value.

The <*PER*>_<*REG*>_<*FIELD*> portion of the constants represents the field name. This is important to the explanation of register configuration macros in section 2.

1.3.3 Bit Field Definition Example

The TMS320DM648 64-Bit Timer Watchdog Timer Control Register (WDTCR), the last member in the register overlay structure shown **Error! Reference source not found.**, has three defined bit fields: WDKEY, WDFLAG, and WDEN. The register configuration header file cslr_tmr.h provides the following definitions relevant to this register:

```
/* WDTCR */

#define CSL_TMR_WDTCR_WDKEY_MASK (0xFFFF0000u)
#define CSL_TMR_WDTCR_WDKEY_SHIFT (0x00000010u)
#define CSL_TMR_WDTCR_WDKEY_RESETVAL (0x00000000u)

/*----WDKEY Tokens----*/
#define CSL_TMR_WDTCR_WDKEY_CMD1 (0x0000A5C6u)
#define CSL_TMR_WDTCR_WDKEY_CMD2 (0x0000DA7Eu)

#define CSL_TMR_WDTCR_WDFLAG_MASK (0x00008000u)
#define CSL_TMR_WDTCR_WDFLAG_SHIFT (0x0000000Fu)
#define CSL_TMR_WDTCR_WDFLAG_RESETVAL (0x00000000u)

/*----WDFLAG Tokens----*/
#define CSL_TMR_WDTCR_WDFLAG_NO_TIMEOUT (0x00000000u)
#define CSL_TMR_WDTCR_WDFLAG_TIMEOUT (0x00000001u)

#define CSL_TMR_WDTCR_WDEN_MASK (0x00004000u)
#define CSL_TMR_WDTCR_WDEN_SHIFT (0x0000000Eu)
#define CSL_TMR_WDTCR_WDEN_RESETVAL (0x00000000u)

/*----WDEN Tokens----*/
#define CSL_TMR_WDTCR_WDEN_DISABLE (0x00000000u)
#define CSL_TMR_WDTCR_WDEN_ENABLE (0x00000001u)

#define CSL_TMR_WDTCR_RESETVAL (0x00000000u)
```

The field names for the WDKEY, WDFLAG and WDEN fields are TMR_WDTCR_WDKEY, TMR_WDTCR_WDFLAG and TMR_WDTCR_WDEN, respectively.

The configuration file also defines tokens for some registers. For example, tokens for the Watchdog Enable (WDEN) field ease enabling and disabling the timer.

2 Macro Reference

CSL_FMK *Field Make*

Macro	CSL_FMK (field, val)
Arguments	field Field name, in the format <PER_REG_FIELD> val Value
Return Value	Uint32
Description	Shifts and AND masks absolute value (val) to specified field location. The result can then be written to the register using the register handle.
Evaluation	`((val) << CSL_##PER_REG_FIELD##_SHIFT) & CSL_##PER_REG_FIELD##_MASK`

Example	To set the Watchdog Timer Enable Bit (WDEN) in the Watchdog Timer Control Register (WDTCR) to ENABLE (1):
	`tmrRegs->WDTCR |= CSL_FMK (TMR_WDTCR_WDEN, 1);`

CSL_FMKT *Field Make Token*

Macro	CSL_FMKT (field, token)
Arguments	field Field name, in the format <PER_REG_FIELD> token Token
Return Value	Uint32
Description	Shifts and AND masks predefined symbolic constant (token) to specified field location (field). The result can then be written to the register using the register handle.
Evaluation	`CSL_FMK(PER_REG_FIELD, CSL_##PER_REG_FIELD##_##TOKEN)`

Example	To set the Watchdog Timer Enable Bit (WDEN) in the Watchdog Timer Control Register (WDTCR) to ENABLE (1):
	`tmrRegs->WDTCR |= CSL_FMKT (TMR_WDTCR_WDEN, ENABLE);`

CSL_FMKR *Field Make Raw*

Macro	CSL_FMKR (msb, lsb, val)
Arguments	msb Most significant bit of field lsb Least significant bit of field val Value
Return Value	Uint32
Description	Shifts and AND masks absolute value (val) to specified field location, specified by raw bit positions representing the most and least significant bits of the field (msb, lsb). The result can then be written to the register using the register handle.
Evaluation	`((val) & ((1 << ((msb) - (lsb) + 1)) - 1)) << (lsb)`

Example	To set the Watchdog Timer Enable Bit (WDEN) in the Watchdog Timer Control Register (WDTCR) to ENABLE (1):
	`tmrRegs->WDTCR |= CSL_FMKR (14, 14, 1);`

CSL_FEXT *Field Extract*

Macro	CSL_FEXT (reg, field)
Arguments	reg Register field Field name, in the format <PER_REG_FIELD>
Return Value	Uint32
Description	Masks bit field (field) of specified register (reg) and right-justifies.
Evaluation	`((reg) & CSL_##PER_REG_FIELD##_MASK) >> CSL_##PER_REG_FIELD##_SHIFT`
Example	Check Timer Global Control Register (TGCR) to see if Timer 1:2 (TIM12RS) is in reset: `if ((CSL_FEXT (tmrRegs->TGCR, TMR_TGCR_TIM12RS))==RESET_ON)…`

CSL_FEXTR *Field Extract Raw*

Macro	CSL_FEXTR (reg, msb, lsb)
Arguments	reg Register msb Most significant bit of field lsb Least significant bit of field
Return Value	Uint32
Description	Masks bit field of register (reg) as specified by raw bit positions representing the most and least significant bits of the field (msb, lsb), and right-justifies.
Evaluation	`((reg) >> (lsb)) & ((1 << ((msb) - (lsb) + 1)) - 1)`
Example	Check Timer Global Control Register (TGCR) to see if Timer 1:2 (TIM12RS) is in reset: `if ((CSL_FEXTR (tmrRegs->TGCR, 0, 0))==RESET_ON) …`

CSL_FINS *Field Insert*

Macro	CSL_FINS (reg, field, val)
Arguments	reg Register field Field name in the format <PER_REG_FIELD> val Value
Return Value	None
Description	Inserts the absolute value (val) at the specified field (field) in the register (reg). This macro modifies the register.
Evaluation	`(reg) = ((reg) & ~CSL_##PER_REG_FIELD##_MASK) | CSL_FMK(PER_REG_FIELD, val)`
Example	Set the Enable Mode for timer 3:4 (ENAMODE34) in the Timer Control Register (TCR) to disabled: `CSL_FINS (tmrRegs->TCR, TMR_TCR_ENAMODE34, 0);`

CSL_FINST *Field Insert Token*

Macro CSL_FINST (reg, field, token)

Arguments

reg	Register
field	Field name, in the format <PER_REG_FIELD>
token	Token

Return Value None

Description Inserts predefined symbolic constant (token) at the specified field (field) in the register (reg). This macro modifies the register.

Evaluation

```
CSL_FINS((reg), PER_REG_FIELD,
CSL_##PER_REG_FIELD##_##TOKEN)
```

Example Set the Enable Mode for timer 3:4 (ENAMODE34) in the Timer Control Register (TCR) to disabled:

```
CSL_FINST (tmrRegs->TCR, TMR_TCR_ENAMODE34, DISABLED);
```

CSL_FINSR *Field Insert Raw*

Macro CSL_FINSR (reg, msb, lsb, val)

Arguments

reg	Register
msb	Most significant bit of field
lsb	Least significant bit of field
val	Value

Return Value None

Description Inserts the absolute value (val) in bit field of register (reg), as specified by raw bit positions representing the most and least significant bits of the field (msb, lsb). This macro modifies the register.

Evaluation

```
(reg) = ((reg)&~(((1<<((msb)-(lsb)+1))-1)<<(lsb)))
    | CSL_FMKR(msb,lsb,val)
```

Example Set the Enable Mode for timer 3:4 (ENAMODE34) in the Timer Control Register (TCR) to disabled:

```
CSL_FINSR (tmrRegs->TCR, 23, 22, 0);
```

3 Examples

This section contains usage examples for the Chip Support Register Configuration Macros. The Platform Support Package (PSP) also provides working examples in the *soc/* <device>*/examples* directory.

3.1 DDR2 Example

This example performs the following steps:

1. Enables the DDR2 module
2. Sets up the hardware to default values and Normal Mode
3. Writes the Invalid values into DDR2 SDRAM area to over write the previous values
4. Writes valid data
5. Does the data comparison to ensure the written data is proper or not
6. Displays the messages based on step 5

For more information on DDR2 registers, see *TMS320DM647/DM648 DSP DDR2 Memory Controller User's Guide* (SPRUEK5A).

```
#include <cslr_ddr2.h>
#include <cslr_sys.h>
#include <stdio.h>
#include <soc.h>
#include <cslr_psc.h>

/** Result - Passed */
#define DATA_MATCH_SUCCESS    1
/** Result - Failed */
#define DATA_MATCH_FAIL       0

/** Data count(number write/readbacks) */
#define DATA_CNT    10

/* Forwards declarations */
static void device_init(void);
static void init_ddr2(void);
static int test_ddr2(void);

/* Pointer to register overlay structure for DDR2 */
CSL_Ddr2RegsOvly ddr2Regs = (CSL_Ddr2RegsOvly)CSL_DDR2_0_REGS;


/*
 * =============================================================================
 *   @func   main
 *
 *   @desc
 *     This is the main routine for the file.
 *
 * =============================================================================
 */
```

```
int main(void)
{
    /* Enable the ddr2 in power sleep controller */
    device_init();

    /* Set up read_write functionality of DDR2 */
    init_ddr2();

       /* Test DDR2 read write functionality, returns 0 for pass and 1 for fail */
    return(test_ddr2());
}

/*
 * =============================================================================
 *   @func   init_ddr2
 *
 *   @desc
 *       This function code showing how to use CSL macros, setting up DDR2
 *
 * =============================================================================
 */
void init_ddr2(void)
{
    Uint32              mask;

    /* Refresh rate*/
    CSL_FINS(ddr2Regs->SDRFC, DDR2_SDRFC_REFRESH_RATE, CSL_DDR2_SDRFC_REFRESH_RATE_RESETVAL);

    /* Writing the ddr2 sdram Settings in SDRAM Config register */
    mask = ~(
              (CSL_DDR2_SDCFG_TIMUNLOCK_MASK) |
              (CSL_DDR2_SDCFG_CL_MASK) |
              (CSL_DDR2_SDCFG_IBANK_MASK) |
              (CSL_DDR2_SDCFG_PAGESIZE_MASK)|
              (CSL_DDR2_SDCFG_NM_MASK));

    /* Set the TIMUNLOCK bit : A write to this bit will cause the DDR2 Memory
                               Controller to start the SDRAM initialization sequence. */
    /* Set the CAS latency of 5 */
    /* Set the Internal SDRAM bank to 4 */
    /* Set the DDR2 data bus width to 32 bit */
    /* Set the page size to 256-word page */

    ddr2Regs->SDCFG = (ddr2Regs->SDCFG & mask ) |
                          (CSL_FMK(DDR2_SDCFG_TIMUNLOCK,
                                   CSL_DDR2_SDCFG_TIMUNLOCK_SET)) |
                          (CSL_FMKT(DDR2_SDCFG_CL, FIVE)) |
                          (CSL_FMKT(DDR2_SDCFG_IBANK, FOUR)) |
                          (CSL_FMKT(DDR2_SDCFG_NM, 32BIT )) |
                          (CSL_FMKT(DDR2_SDCFG_PAGESIZE, 256W_PAGE ));

    /* Unlock the BOOT_UNLOCK bit */
    CSL_FINS(ddr2Regs->SDCFG, DDR2_SDCFG_BOOT_UNLOCK, \
                                      CSL_DDR2_SDCFG_BOOT_UNLOCK_UNLOCKED);
    /* Set the DDR2 SDRAM drive strength to Normal */
    CSL_FINS(ddr2Regs->SDCFG, DDR2_SDCFG_DDR_DRIVE, \
                                      CSL_DDR2_SDCFG_DDR_DRIVE_NORMAL);
    /* Lock the BOOT_UNLOCK bit */
    CSL_FINS(ddr2Regs->SDCFG, DDR2_SDCFG_BOOT_UNLOCK, \
                                      CSL_DDR2_SDCFG_BOOT_UNLOCK_LOCKED);
```

```
   /* Locking the timing_unlock to prevent further changes */
    CSL_FINS(ddr2Regs->SDCFG, DDR2_SDCFG_TIMUNLOCK,
                                      CSL_DDR2_SDCFG_TIMUNLOCK_CLEAR);
}

/*
 * ============================================================================
 *   @func   test_ddr2
 *
 *   @desc
 *       example code showing DDR2 read write functionality test
 *
 * ============================================================================
 */
int test_ddr2(void)
{
    volatile Uint32     result, index ;
    Uint32              tempData;

       /* Pointer that points to SDRAM start area */
    Uint32 *pDdr2Data = (Uint32 *)CSL_DDR2_SDRAM_ADDR ;

    printf("\n Testing DDR2 read write functionality\n");

       /* Write 'invalid' values into DDR2 SDRAM area. This is to overwrite the
    * previous valid values
    */
    tempData = 0xdeadbeef;
    for (index = 0; index < DATA_CNT; index++) {
        pDdr2Data[index] =  tempData;
    }

    /* Write **valid** values into SDRAM area. */
    tempData = 0x56780000;
    for (index = 0; index < DATA_CNT; index++) {
        pDdr2Data[index] = tempData + index ;
    }

    /* Verify that the data was indeed written */
    result = DATA_MATCH_SUCCESS;
    for (index = 0; index < DATA_CNT; index++) {
        if (pDdr2Data[index] != (tempData + index)) {
            result = DATA_MATCH_FAIL;
            break ;
        }
    }

   /* Print the appropriate message based on result */
    if (result == DATA_MATCH_SUCCESS) {
        printf("\n Write to and Read from DDR2 SDRAM is Successful\n");
             printf("\n DDR2 read write example passed\n");
             return(0);
    }
    else {
        printf("\n Write to and Read from DDR2 SDRAM is NOT Successful\n");
             printf("\n DDR2 read write example failed\n");
             return(1);
    }
}

/*
```

```
 * ============================================================================
 *   @func   device_init
 *
 *   @desc
 *     This function enables the DDR2 in the power and sleep controller.
 *
 *  @arg
 *      None
 *
 *  @return
 *      None
 * ============================================================================
*/
void device_init(void)
{

  CSL_PscRegsOvly pscRegs = (CSL_PscRegsOvly)CSL_PSC_0_REGS;

  // deassert DDR2 local PSC reset and set NEXT state to ENABLE
  pscRegs->MDCTL[CSL_PSC_DDR2] = CSL_FMKT(PSC_MDCTL_NEXT, ENABLE) |
                                  CSL_FMKT(PSC_MDCTL_LRST, DEASSERT);
  //move DDR2 PSC to Next state
  pscRegs->PTCMD = CSL_FMKT(PSC_PTCMD_GO0, SET);

  //wait for transition
  while (CSL_FEXT( pscRegs->PTSTAT, PSC_PTSTAT_GOSTAT0)
          == CSL_PSC_PTSTAT_GOSTAT0_IN_PROGRESS);
}
```

3.2 PLLC Example

The given example describes the delay routine, main routine which calls example routine, actual routine which configures the PLL0, and the dummy function. For more information, see the *TMS320DM647/DM648 DSP Subsystem Reference Guide* (SPRUEU6).

```
#include <stdio.h>
#include <cslr_pllc1.h>
#include <soc.h>

#define CSL_PLLC1_PLLCTL_PLLENSRC_REGBIT  (0x00000000u)
#define CSL_PLLC1_PLLCTL_PLLRST_NO        (0x00000000u)
#define CSL_PLLC1_PLLCTL_PLLRST_YES       (0x00000001u)

static void setupPll1(int pll_multiplier);
static int test_pll1();

/* Pointer to register overlay structure */
CSL_Pllc1RegsOvly pllcRegs = ((CSL_Pllc1RegsOvly)CSL_PLLC_1_REGS);

/*
 * ============================================================================
 *   @func   sw_wait
 *
 *   @desc
 *     This is the delay routine.
 *
 * ============================================================================
 */
void sw_wait(int delay)
```

```
{
    volatile int i;
    for( i = 0; i < delay; i++ ) {
    }
}

/*
 * =============================================================================
 *   @func   main
 *
 *   @desc
 *     This is the main routine which calls example routine.
 *
 * =============================================================================
 */

int main()
{
       printf("Configure PLL1 with register layer macros\n");
       printf("Please wait System PLL Initialization is in Progress.....\n");

       return(test_pll1());
}

/*
 * =============================================================================
 *   @func   setupPll1
 *
 *   @desc
 *     This is the actual routine which configures PLL0.
 *
 * =============================================================================
 */
void setupPll1(int pll_multiplier)
{
       /* Set PLLENSRC '0', PLL Enable(PLLEN) selection is controlled through MMR */
    CSL_FINST(pllcRegs->PLLCTL, PLLC1_PLLCTL_PLLENSRC, REGBIT);

    /*Set PLL BYPASS MODE */
    CSL_FINST(pllcRegs->PLLCTL, PLLC1_PLLCTL_PLLEN, BYPASS);

    /*wait for some cycles to allow PLLEN mux switches properly to bypass clock*/
    sw_wait(150);

    /* Reset the PLL */
    CSL_FINST(pllcRegs->PLLCTL, PLLC1_PLLCTL_PLLRST, YES);

    /*PLL stabilisation time*/
    sw_wait(1500);

    /*Program PREDIV Reg, POSTDIV register and OSCDIV1 Reg
    1.predvien_pi is set to '1'
    2.prediv_ratio_lock_pi is set to '1', RATIO field of PREDIV is locked
    3.Set the PLLM Register
    4.Dont program POSTDIV Register
    */

    /* Set PLL Multiplier */
    pllcRegs->PLLM = pll_multiplier;

    /*wait for PLL to Reset properly=>PLL reset Time*/
    sw_wait(128);

    /*Bring PLL out of Reset*/
    CSL_FINST(pllcRegs->PLLCTL, PLLC1_PLLCTL_PLLRST, NO);

    /*Wait for PLL to LOCK atleast 2000 MXI clock or Reference clock cycles*/
    sw_wait(2000);

    /*Enable the PLL Bit of PLLCTL*/
    CSL_FINST(pllcRegs->PLLCTL, PLLC1_PLLCTL_PLLEN, PLL);
}

/*
 * ============================================================================
 *   @func   test_pll0
 *
 *   @desc
 *     This is the dummy function.
 *
 * ============================================================================
 */
int test_pll1()
{
      setupPll1(20);

      printf("PLL1 has been configured\n");

      return(0);
}
```

3.3 GPIO Example

This example sets up GPIO pin as an output and a GPIO pin as an input using the register layer CSL. The example is bios based and will toggle GPIO output pin to cause a rising edge transition to occur on the GPIO input pin. The rising edge transition detected by GPIO input pin will cause an interrupt to occur. The ISR will set a pass/fail flag which is passed by the SYS_exit api to the exit function. Open the bios message log to view the test results. For more information on GPIO registers, see *TMS320DM647/DM648 DSP General-Purpose Input/Output (GPIO) User's Guide* (SPRUEK7A).

```
#include <log.h>
#include <tsk.h>
#include <c64.h>
#include <stdlib.h>
#include <soc.h>
#include <cslr_gpio.h>
#include <cslr_psc.h>
#include <cslr_sys.h>
#include <hwi.h>
#include "gpio_interruptcfg.h"

#define LOW 0x0
#define HIGH 0x1

static void device_init(void);
static void setup_GPIO();
void myExit(int);
void test_GPIO(void);
void GPIO_input_isr();

static int status=1;                         // Pass/Fail flag

CSL_GpioRegsOvly gpioRegs = (CSL_GpioRegsOvly)CSL_GPIO_0_REGS;

void main (void)
{
  // Enable GPIO in the power and sleep controller
  device_init();

  // Configure output and input GPIO, enable rising edge detection
  setup_GPIO();

}

void setup_GPIO()
{
  // The setup code below uses the raw register layer macros to allow the pin
  // numbers to be passed in via pre-defined symbols in the pjt settings.  If
  // variable pin settings are not required, the token macros should be used.

  // Configure GPIO(OUTPUT_PIN) as an output and GPIO(INPUT_PIN) as an input
  gpioRegs->DIR01 = CSL_FMKR(OUTPUT_PIN,OUTPUT_PIN,LOW)
                  | CSL_FMKR(INPUT_PIN,INPUT_PIN,HIGH);

  // Set Data low in SET_DATA register for GPIO(OUTPUT_PIN)
  gpioRegs->CLR_DATA01 = CSL_FMKR(OUTPUT_PIN,OUTPUT_PIN,HIGH);

  // Enable GPIO Bank 0 interrupts
  CSL_FINST(gpioRegs->BINTEN,GPIO_BINTEN_EN0,ENABLE);

  // Configure GPIO(INPUT_PIN) to generate interrupt on rising edge
  CSL_FINSR(gpioRegs->SET_RIS_TRIG01,INPUT_PIN,INPUT_PIN,HIGH);

}

void test_GPIO(void)
{
  // Enable cpu interrupt 4 in IER
  C64_enableIER(C64_EINT4);

  LOG_printf(&trace, "GPIO Interrupt Test.");

  LOG_printf(&trace, "Set output GPIO high.");

  // Set GPIO(OUTPUT_PIN) high
  gpioRegs->SET_DATA01 = CSL_FMKR(OUTPUT_PIN,OUTPUT_PIN,HIGH);

  // Wait for GPIO(OUTPUT_PIN) to go high
  while(CSL_FEXTR(gpioRegs->OUT_DATA01,OUTPUT_PIN,OUTPUT_PIN)!=1);

  // Call exit function
  SYS_exit(status);
}

void GPIO_input_isr()
{
  LOG_printf(&trace, "GPIO interrupt occured.");
  // Set flag to 0 indicating ISR occurred
  status=0;
}

void myExit(int test_result)
{
  // Turn off interrupts
  HWI_disable();

  if(test_result==0)
    LOG_printf(&trace, "GPIO interrupt test: PASSED.");
  else
    LOG_printf(&trace, "GPIO interrupt test: FAILED.  Check jumper.");

  exit(0);
}

void device_init(void)
{
  CSL_SysRegsOvly sysRegs = (CSL_SysRegsOvly)CSL_SYS_0_REGS;
  CSL_PscRegsOvly pscRegs = (CSL_PscRegsOvly)CSL_PSC_0_REGS;

  // mux between Timer 0/1 and GPIO[8:11], enable GPIO[8:11]
  sysRegs->PINMUX = CSL_FMKT(SYS_PINMUX_TIMER_EN,TIM0_TIM1_DISABLE);

  // deassert GPIO local PSC reset and set NEXT state to ENABLE
  pscRegs->MDCTL[CSL_PSC_GPIO] = CSL_FMKT( PSC_MDCTL_NEXT, ENABLE )
                               | CSL_FMKT( PSC_MDCTL_LRST, DEASSERT );
  // move GPIO PSC to Next state
  pscRegs->PTCMD = CSL_FMKT(  PSC_PTCMD_GO0, SET );

  // wait for transition
  while ( CSL_FEXT( pscRegs->MDSTAT[CSL_PSC_GPIO], PSC_MDSTAT_STATE )
          != CSL_PSC_MDSTAT_STATE_ENABLE );
}
```

4 References

- *TMS320DM648 Digital Media Processor (DMP) Datasheet* (SPRS372)
- *TMS320DM647/DM648 DSP DDR2 Memory Controller User's Guide* (SPRUEK5A).
- *TMS320DM647/DM648 DSP Subsystem Reference Guide* (SPRUEU6).
- *TMS320DM647/DM648 DSP General-Purpose Input/Output (GPIO) User's Guide* (SPRUEK7A)