Application Report
SPRAAK7 – July 2007

Using the Chip Support Register Configuration Macros

Platform Support Group

ABSTRACT

This document describes the Chip Support Register Configuration files provided for some DaVinci 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

The Chip Support Register Configuration files provide register configuration support for each of the peripheral modules on selected DaVinci 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. Please see Table 1 for a list of supported devices.

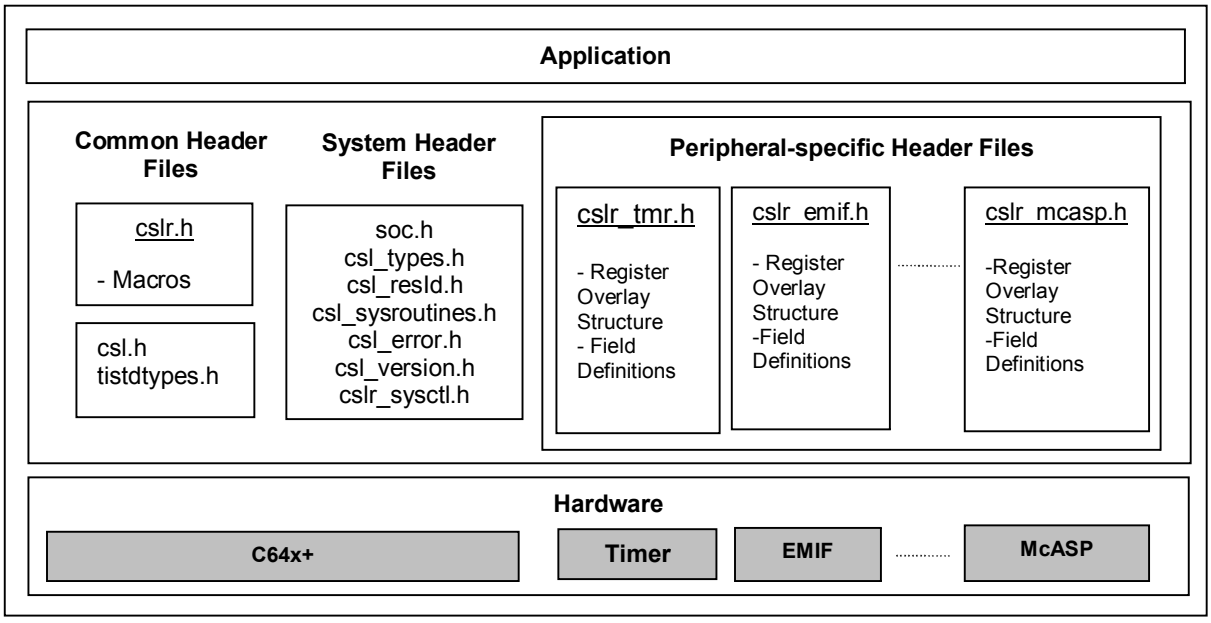
Family	Devices	Delivery Mechanism
TMS320DM643x DMP	DM6431, DM6433, DM6435, DM6437	DSP/BIOS PSP for the DM6437 Digital Video Development Platform (DVDP)
TMS320C642x DSP	C6424, C6421	DSP/BIOS PSP for C6424 EVM

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 DM643x devices shown in Table 1 are supported by the header files in the directory */soc/dm6437/dsp/inc* in the DM6437 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 that some peripherals are made up of multiple header file components. For example, the TMS320DM643x video processing subsystem (VPSS) is split into multiple modules consisting of a set of front end and back end components.

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, VPSS clock, 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: VPSS clock, 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 DM6437 64-Bit Timer, from the file cslr_tmr.h:

```
/**************************************************************************\
* Register Overlay Structure
\**************************************************************************/
typedef struct  {
    volatile Uint32 PID12;
    volatile Uint32 EMUMGT;
    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;
```

Please refer to the TMS320DM643x DMP 64-Bit Timer User's Guide (SPRU989) section on registers for further information on the timer registers.

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 TMS320DM6437 64-Bit Timer Watchdog Timer Control Register (WDTCR), the last member in the register overlay structure shown above, 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 Asynchronous EMIF Example

This example sets up the DM6437 Asynchronous EMIF control registers using register configuration macros, according to the 'bit size' input parameter. This example assumes that the pin multiplex settings are already configured for 8- or 16-bit EMIF and that the Power Sleep Controller (PSC) is initialized for EMIF operation. Please see the *TMS320DM643x DMP External Memory Interface (EMIF) User's Guide* (SPRU984) for descriptions of EMIF registers.

```
#include <csl_types.h>
#include <cslr.h>
#include <soc.h>
#include <cslr_emif.h>
#include <csl_error.h>


CSL_Status SetupEmifA(int bus_width)
{
  CSL_EmifRegsOvly emifRegs = (CSL_EmifRegsOvly)CSL_EMIFA_0_REGS; // Pointer to EMIF register
                                                                  // overlay structure
  CSL_Status status = CSL_SOK;

  emifRegs = ((CSL_EmifRegsOvly)CSL_EMIFA_0_REGS);

  // No extended wait cycles
  emifRegs->AWCCR = CSL_FMKT(EMIF_AWCCR_WP0, WAITLOW) |
                    CSL_FMK(EMIF_AWCCR_MEWC, 0);

  if(bus_width == 8)  {                              // Set up for 8-bit bus
    emifRegs->A1CR = ( CSL_FMK(EMIF_A1CR_SS, 0) |
                       CSL_FMK(EMIF_A1CR_EW, 0) |
                       CSL_FMK(EMIF_A1CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A1CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A1CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A1CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A1CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A1CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A1CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A1CR_ASIZE, ASIZE_8BITS));

    emifRegs->A2CR = ( CSL_FMK(EMIF_A2CR_SS, 0));
                       CSL_FMK(EMIF_A2CR_EW, 0) |
                       CSL_FMK(EMIF_A2CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A2CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A2CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A2CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A2CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A2CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A2CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A2CR_ASIZE, ASIZE_8BITS));
```

```
    emifRegs->A3CR = ( CSL_FMK(EMIF_A3CR_SS, 0));
                       CSL_FMK(EMIF_A3CR_EW, 0) |
                       CSL_FMK(EMIF_A3CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A3CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A3CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A3CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A3CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A3CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A3CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A3CR_ASIZE, ASIZE_8BITS));

    emifRegs->A4CR = ( CSL_FMK(EMIF_A4CR_SS, 0));
                       CSL_FMK(EMIF_A4CR_EW, 0) |
                       CSL_FMK(EMIF_A4CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A4CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A4CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A4CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A4CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A4CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A4CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A4CR_ASIZE, ASIZE_8BITS));
  }
  elseif (busWidth == 16)  {                                  // Set up for 16-bit bus
    emifRegs->A1CR = ( CSL_FMK(EMIF_A1CR_SS, 0) |
                       CSL_FMK(EMIF_A1CR_EW, 0) |
                       CSL_FMK(EMIF_A1CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A1CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A1CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A1CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A1CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A1CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A1CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A1CR_ASIZE, ASIZE_16BITS));

    emifRegs->A2CR = ( CSL_FMK(EMIF_A2CR_SS, 0));
                       CSL_FMK(EMIF_A2CR_EW, 0) |
                       CSL_FMK(EMIF_A2CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A2CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A2CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A2CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A2CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A2CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A2CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A2CR_ASIZE, ASIZE_16BITS));

    emifRegs->A3CR = ( CSL_FMK(EMIF_A3CR_SS, 0));
                       CSL_FMK(EMIF_A3CR_EW, 0) |
                       CSL_FMK(EMIF_A3CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A3CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A3CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A3CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A3CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A3CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A3CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A3CR_ASIZE, ASIZE_16BITS));
```

```
    emifRegs->A4CR = ( CSL_FMK(EMIF_A4CR_SS, 0));
                       CSL_FMK(EMIF_A4CR_EW, 0) |
                       CSL_FMK(EMIF_A4CR_W_SETUP, 0xf) |
                       CSL_FMK(EMIF_A4CR_W_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A4CR_W_HOLD, 0x7) |
                       CSL_FMK(EMIF_A4CR_R_SETUP, 0xf) |
                       CSL_FMK(EMIF_A4CR_R_STROBE, 0x3f) |
                       CSL_FMK(EMIF_A4CR_R_HOLD, 0x7) |
                       CSL_FMK(EMIF_A4CR_TA, 0x3) |
                       CSL_FMKT(EMIF_A4CR_ASIZE, ASIZE_16BITS));
  }
  else  {
    status = CSL_ESYS_INVPARAMS;                          // Invalid bus size
  }

  emifRegs->NANDFCR = 0x00000000;                         // NAND controller not used

  return status;
}
```

3.2 PLL1 Setup Example

This example configures DM6437 PLL1 control registers based on clock source and PLL multiplier input using register configuration macros. This example assumes that the Power Sleep Controller (PSC) is already initialized for PLL operation. Please see the *TMS320DM643x DMP DSP Subsystem Reference Guide* (SPRU978) for PLL register descriptions .

```
#include <csl_types.h>
#include <cslr.h>
#include <soc.h>
#include <cslr_pll1.h>


void SetupPll1(int clock_source, int pll_multiplier)
{
  CSL_Pll1RegsOvly pll1Regs = (CSL_Pll1RegsOvly)CSL_PLLC_0_REGS;  // Pointer to register
                                                                  // overlay structure

  /* Set PLL to bypass mode */
  CSL_FINS(pll1Regs->PLLCTL, PLL1_PLLCTL_CLKMODE, 0);         // Clear clock source mode

  if (clock_source == CSL_PLL1_PLLCTL_CLKMODE_OSCIN)
    CSL_FINST(pll1Regs->PLLCTL, PLL1_PLLCTL_CLKMODE, OSCIN);  // Set clock source to
oscillator
  else
    CSL_FINST(pll1Regs->PLLCTL, PLL1_PLLCTL_CLKMODE, CLKIN);  // Set source to ext clock

  CSL_FINST(pll1Regs->PLLCTL, PLL1_PLLCTL_PLLEN, BYPASS_MODE);    // Set PLL to Bypass mode

  sw_wait(150);                                               // Wait for bypass mode switch

  CSL_FINST (pll1Regs->PLLCTL, PLL1_PLLCTL_PLLRST, IN_RESET);     // Reset PLL
  CSL_FINST (pll1Regs->PLLCTL, PLL1_PLLCTL_PLLDIS, DISABLE);      // Disable PLL
  CSL_FINST (pll1Regs->PLLCTL, PLL1_PLLCTL_PLLPWRDN, POWERED_UP); // Power up PLL
  CSL_FINST (pll1Regs->PLLCTL, PLL1_PLLCTL_PLLDIS, ENABLE);       // Enable PLL
```

```
  /* Set PLL Multiplier */
  pll1Regs->PLLM = pll_multiplier;

  sw_wait(150);                                                  // Wait for PLL to lock

  CSL_FINST(pll1Regs->PLLCTL, PLL1_PLLCTL_PLLRST, NOT_IN_RESET);  // Release PLL from Reset

  sw_wait(2000);                                                 // Wait for PLL to lock

  CSL_FINST(pll1Regs->PLLCTL, PLL1_PLLCTL_PLLEN, PLL_MODE);       // Set PLL to PLL mode
}

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

3.3 GPIO Example

This example initializes the Power Sleep Controller (PSC) for GPIO operation, configures GPIO pins, and reads and writes GPIO data. Please see the *TMS320DM643x DMP General-Purpose Input/Output (GPIO) User's Guide* (SPRU988) for GPIO register descriptions.

```
#include <csl_types.h>
#include <soc.h>
#include <cslr_sysctl.h>
#include <cslr_psc.h>
#include <cslr_gpio.h>
#include <stdio.h>

void test_GPIO (void) {
  CSL_PscRegsOvly pscRegs = (CSL_PscRegsOvly)CSL_PSC_0_REGS;
  CSL_GpioRegsOvly gpioRegs = (CSL_GpioRegsOvly)CSL_GPIO_0_REGS;

  // 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_G00, SET );

  // Wait for transition to NEXT state
  while ( CSL_FEXT( pscRegs->MDSTAT[CSL_PSC_GPIO], PSC_MDSTAT_STATE )
          != CSL_PSC_MDSTAT_STATE_ENABLE );

  // Configure pin 1 as an output, then set the data High (Low->High).
  CSL_FINS( gpioRegs->DIR01, GPIO_DIR01_DIR1, CSL_GPIO_DIR01_DIR1_OUT );

  // Set Data high in SET_DATA register
  gpioRegs->SET_DATA01 =
      CSL_FMK( GPIO_SET_DATA01_SET1, CSL_GPIO_SET_DATA01_SET1_SET );

  sw_wait( 1 );

  // read IN_DATA
  if ( CSL_FEXT( gpioRegs->IN_DATA01, GPIO_IN_DATA01_IN1 ) != CSL_GPIO_IN_DATA01_IN1_SET ) {
       printf("Driving GPIO1 high failed!\n" );
```

```
        return;
    }

    sw_wait( 1 );

    // Set Data low in CLR_DATA register
    gpioRegs->CLR_DATA01 =
        CSL_FMK( GPIO_CLR_DATA01_CLR1, CSL_GPIO_CLR_DATA01_CLR1_CLR );

    sw_wait( 1 );

    // read IN_DATA
    if ( CSL_FEXT( gpioRegs->IN_DATA01, GPIO_IN_DATA01_IN1 )
         != CSL_GPIO_IN_DATA01_IN1_CLR ) {
        printf("Driving GPIO1 low failed!\n" );
        return;
    }

    printf( "Test of GPIO1 is successful!\n" );
}

// delay loop
void sw_wait( int delay ) {
    int i;
    for( i = 0 ; i < delay ; i++ ) {
    }
}
```

4 References

- *TMS320DM6437 Digital Media Processor (DMP) Datasheet* (SPRS345)
- *TMS320DM643x DMP DSP Subsystem Reference Guide* (SPRU978)
- *TMS320DM643x DMP 64-Bit Timer User's Guide* (SPRU989)
- *TMS320DM643x DMP External Memory Interface (EMIF) User's Guide* (SPRU984)
- *TMS320DM643x DMP General-Purpose Input/Output (GPIO) User's Guide* (SPRU988)
- *TMS32C6424 Fixed Point Digital Signal Processor Datasheet* (SPRS347)