Texas Instruments

Application Report
SPRAC22–March 2016

ADAS Power Management

Piyali Goswami, Stanley Liu, Chetan Matad, and Sushaanth Srirangapathi *ADAS Software, Processor BU*

ABSTRACT

Power Management (PM) in Advanced Driver Assist Systems (ADAS) requires setting the right power and clock configurations that allow any IP to consume optimal power. This helps not only reduce the total power consumed by the device, but also manage thermal dissipation of the silicon. This application report looks at different ways in which power in the TDA2xx, TDA2ex and TDA3xx family of devices can be managed, and the software APIs to achieve the same.

Contents

List of Figures

List of Tables

1 PRCM Hardware – An Introduction

The Power Reset Clock and Module (PRCM) power management hardware is built with four levels of resource management: module, clock, power, and voltage. These management levels are enforced by defining the managed entities or building blocks of the power management architecture, called the clock, power, and voltage domains. A domain is a group of modules or subsections of the device that share a common entity (for example, common clock source, common voltage source, or common power switch). Figure 1 shows the different PRCM levels of resource management in the TDA2xx/TDA2ex and TDA3xx family of devices.

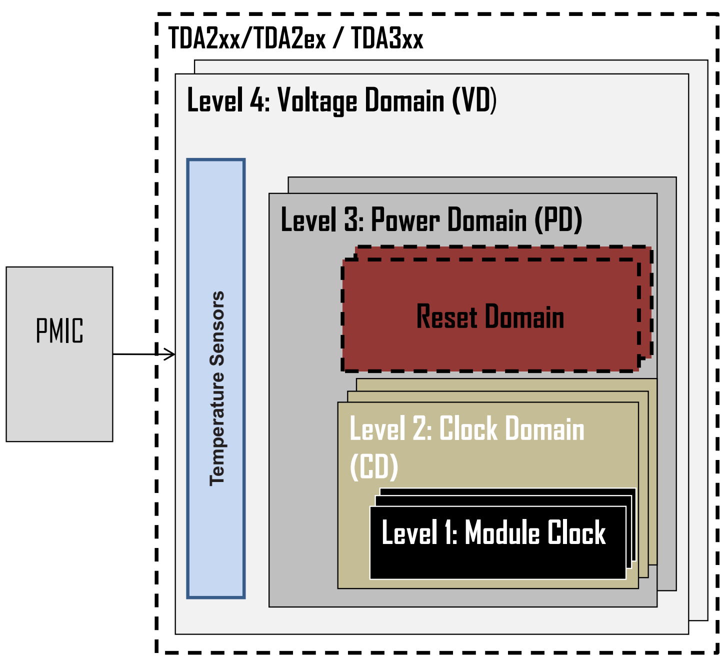


Figure 1. PRCM Power Management Levels

There are four levels of power management:

- **Level 1** is at the module level where certain module clocks may be active when operating in specific modes, or they may be gated. They are typically controlled by PRCM registers such as CM_<*>_CLKCTRL.
- **Level 2** is at the clock domain level where dynamic power consumption is controlled. By gating the clocks in a clock domain, the clocks to all the modules belonging to that clock domain can be cut to lower their active power consumption (that is, the device is on and the clocks to the modules are dynamically switched to ACTIVE or INACTIVE [gated] state). Clock domain behavior can be controlled by CM_<*>_CLKSTCTRL registers.
- **Level 3** is at the power domain level where leakage power consumption can be lowered by turning off or by entering a retention state for a given power domain without affecting the other parts of the device. Power domains of the device can be controlled by PM_<*>_PWRSTCTRL registers and their status can be read from PM_<*>_PWRSTST registers.

  Additionally a power domain also contains reset domains. Reset domains are a group of modules that share a reset line. The modules within a given reset domain can have their reset lines asserted or de-asserted using the RM_<*>_RSTCTRL register of the reset domain. The corresponding status of the reset signal can be read from the RM_<*>_RSTST register.
- **Level 4** is at the voltage domain level. A voltage domain is a section of the device supplied by a dedicated voltage source (that is, an internal LDOs or external switch mode power supply [SMPS]). The software can optimally configure the domain voltage levels to specific values with in the operational voltage range of the device by reading Efuse values specific to the silicon sample.

The TDA2xx/TDA2ex/TDA3xx device also has on-chip thermal sensors that sense the junction temperature of the device and convert it to a 10-bit ADC value. This value can then be used to perform thermal management of the device through software.

Some key differences between the TDA2xx/TDA2ex and TDA3xx PRCM architecture are highlighted in Table 1.

Table 1. Key Differences Between PRCM Architecture of TDA2xx/TDA2ex and TDA3xx

PRCM Feature	TDA2xx/TDA2ex	TDA3xx
Number of Voltage Domains	Five voltage domains (VD_CORE, VD_MPU, VD_DSPEVE, VD_GPU and VD_IVA)	Two voltage domains (VD_CORE, VD_DSPEVE).
Temperature Sensors	Five (one in each voltage domain)	One (in VD_CORE)
Adaptive Body Bias	Supported	Not Supported
Number of DPLLs	13 + 2 (1) Video PLLs	5

(1) TDA2ex has only 1 Video PLL.

The number of DPLLs are different between TDA2xx/TDA2ex and TDA3xx, making the clocking architecture between TDA2xx/TDA2ex and TDA3xx different.

For more details on the PRCM subsystem and its components, see the *Power, Reset, and Clock Management* section in the *TDA2x ADAS Applications Processor Technical Reference Manual* (SPRUHK5) [5] and the *TDA3x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.0 Technical Reference Manual* (SPRUHQ7) .

The following sections discuss the ways in which the power consumption of the device and the thermal dissipation of the device can be kept in check. Section 2 describes how the system can be initialized by setting the right power modes for different modules. Section 3 describes how the modules can be sourced with appropriate clock frequencies and voltage to allow different portions of the device work at the appropriate OPP (Operating Performance Point) taking into account process variations of the device samples. Section 4 describes the method of dynamic power management of the CPUs, which allows the CPU to be in low-power mode when the CPU has nothing to do and then wake up from the low-power mode when it has to resume processing. Section 5 describes a thermal management technique where once the device temperature has become too high the software can configure the use case parameters to allow the device to cool down and then again resume normal use case operation.

2 PM Software Stack

Figure 2 provides the top level view of the power management software stack. Essentially, the software stack is divided in to two layers: PMHAL and PMLIB.

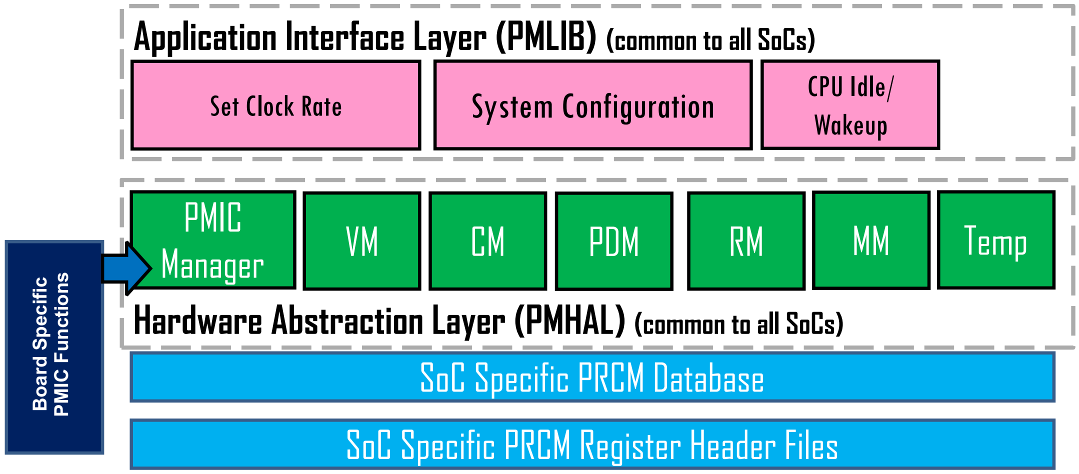


Figure 2. Starterware (STW) Power Management Software Stack

PM Hardware Abstraction Layer (PMHAL) provides low level APIs that allow:

- Programming PRCM registers
  - Power Domain Manager (PDM)
    `<STW Install Dir>\include\pm\pmhal\pmhal_pdm.h`
  - Clock Domain Manager (CM)
    `<STW Install Dir>\include\pm\pmhal\pmhal_cm.h`
  - Reset Manager (RM)
    `<STW Install Dir>\include\pm\pmhal\pmhal_rm.h`
  - Module Manager (MM)
    `<STW Install Dir>\include\pm\pmhal\pmhal_mm.h`
- Programming Temperature Sensor Registers (Temp)
  `<STW Install Dir>\include\pm\pmhal\pmhal_bgap.h`
- Programming Voltage Domain Adaptive Voltage Scaling (AVS) and Adaptive Body Bias (ABB) (VM)
  `<STW Install Dir>\include\pm\pmhal\pmhal_vm.h`
- Programming board specific Power Management IC (PMIC)
  `<STW Install Dir>\include\pm\pmhal\pmhal_pmic.h`
  - Example implementations for TPS659039, TPS65917, L8731, and so forth:
    `<STW Install Dir>\pm\pmhal\pmhal_tps659039.c, pmhal_tps65917.c, pmhal_l8731.c`

PM Library (PMLIB) provides application interface to:

- Configure system level clock frequencies (Section 3)
  `<STW Install Dir>\include\pm\pmlib\pmlib_clkrate.h` and `pmlib_videopll.h`
- Configure the system power states (Section 4)
  `<STW Install Dir>\include\pm\pmlib\pmlib_sysconfig.h`
- Perform dynamic CPU power optimization (Section 5)
  `<STW Install Dir>\include\pm\pmlib\pmlib_cpuidle.h`

3 System Clock Frequency and Voltage Initialization

The initialization of the device to run a given use case involves setting the right frequency for the clocks for the CPUs and peripherals. Along with the frequency setting, the right voltage for the voltage rails is also important to ensure the leakage is kept minimal and the silicon performance is kept optimal.

3.1 Setting the Voltage at Boot

Setting the voltage requires setting the right AVS voltage for the given voltage rail of the given frequency of operation for the CPU in that voltage domain.

The basic mechanism of AVS Class 0 is described in Figure 3.

- For a given silicon sample, the minimum voltage is determined at which the device is operational; some of the voltage margin is added to the minimum voltage.
- This voltage value is fused into the device registers readable from the software.
- This voltage value is then programmed into the PMIC via the I2C instance on the device.
- Once programmed, the PMIC can provide the required voltage on the device voltage rails.

The advantage of using the AVS Class-0 is to ensure that the voltages supplied to the device are adaptable to the process variations of the device and the power consumption is kept optimal.

For more details on AVS voltages and I2C to program, see the *TDA2x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 2.0, 1.x Technical Reference Manual* (SPRUHK5) [2], *TDA3x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.0 Technical Reference Manual* (SPRUHQ7) [3], *TDA2x ADAS Application Processor 23mm Package (ABC Package) Processor Data Manual* (SPRS859) [6], and the *TDA3x ADAS Applications Processor 15mm Package (ABF Package) Data Manual* (SPRS916) [7].

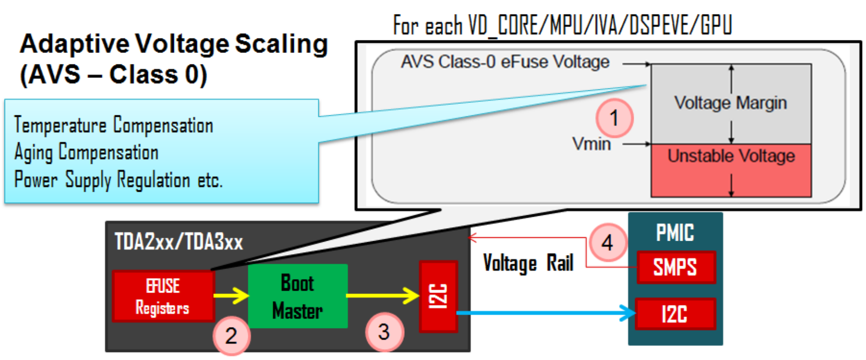


Figure 3. AVS Class 0 Voltage Scaling Mechanism

On system boot, before the DPLLs are configured, the voltage rails should be set to the required voltages as per AVS Class 0 in the SBL. This ensures hot devices do not enter a thermal condition on boot. Setting the AVS Class 0 ensures silicon reliability and ensured lifetime POHs are achieved.

The TDA2xx/TDA2ex family of devices also support Adaptive Body Bias (ABB), which helps control further leakage, improve the performance of the silicon by applying a voltage to the NWELL of the PMOS transistors of the device, and change V_{TH} of the transistors.

Table 2. Adaptive Body Bias (ABB) Impact on Strong and Weak Samples

Reverse Body Bias (RBB)	Forward Body Bias (FBB)
VBBNW > VDD	VBBNW < VDD
For Strong Samples	For Weak Samples
Increase V_{th}	Decrease V_{th}
Reduce Leakage	Increase Performance

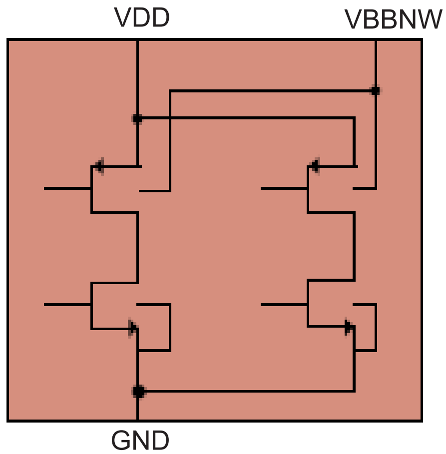


Figure 4. VDD and VBBNW Supply to Device Transistors

There are given values of AVS-Class 0 and ABB LDO voltage for each OPP. The PMHAL provides APIs to program the voltage rails with AVS and ABB (if available) for a given OPP (OPP_LOW/OPP_NOM/OPP_OD/OPP_HIGH).

The API sequence during boot, to set the AVS and ABB voltages for voltage domain VD_DSPEVE, is shown in Figure 6. Note the board specific PMIC operation functions are registered with the PMHAL PMIC Manager by the application before the Voltage Manager API is called. This is done to ensure the use of the same software across multiple boards using different PMIC solutions. STW PM HAL provides sample implementations of the PMIC functions corresponding to TI TPS659039Q, TPS65917, LP8731, and so forth.

```
#include "pmlib_sysconfig.h"
#include "pmhal_vm.h"
#include "pmhal_pmic.h"
#include "pmhal_tps65917.h"

pmlibSysConfigPowerStateParams_t inputTable[] =
{{PMHAL_PRCM_MOD_I2C1, PMLIB_SYS_CONFIG_ALWAYS_ENABLED}};
pmhalVmOppId_t  oppId;
const pmhalPmicOperations_t *pmicOps;

retVal = PMLIBSysConfigSetPowerState(inputTable, 1U, PM_TIMEOUT_INFINITE, NULL);
if (PM_SUCCESS == retVal)
{
    /* Get the pmic ops and register with the pmic interface. */
    pmicOps = PMHALTps65917GetPMICOps();
    retVal = PMHALPmicRegister(pmicOps);
    /* Configure AVS and ABB */
    retVal |= PMHALVMSetOpp(PMHAL_PRCM_VD_DSPEVE,
              PMHAL_VM_OPP_NOM,
              PM_TIMEOUT_INFINITE);
}
```

3.2 Taking Care of Differences in Device Voltage Rail and PMIC Integration

The PMIC integration with the device on different board designs can differ in the following three aspects:

- Different PMIC regulator outputs (LDO, BUCK, SMPS) to different input device voltage rails
- Different I2C instance of the device is used to communicate with the PMIC
- Different PMIC I2C slave address as determined by the PMIC One Time Programmed (OTP) code.

In order to take care of these differences each PMHAL PMIC driver provides an API to register the mapping, device I2C instance and I2C slave address with the driver in case these parameters differ from the default configuration provided. Note the default configurations correspond to the TI EVMs. The API for the LP8731 PMIC driver is as shown below:

```
/**
 * \brief   The PMIC regulator output to the device input mapping can be
 *          different on different boards. This API can be used to provide
 *          a different mapping to the PMIC driver if the mapping does not
 *          match the default. Example table is shown below:
 *       --------------------------------------------------------------------
 *       | Device voltage Rail            | Ptr to Regulator                |
 *       --------------------------------------------------------------------
 *       | PMHAL_PRCM_PMIC_REGULATOR_CORE | PMHAL_LP8731_REGULATOR_BUCK1    |
 *       | PMHAL_PRCM_PMIC_REGULATOR_DSPEVE| PMHAL_LP8731_REGULATOR_BUCK2   |
 *       | ....                                                             |
 *       | index (Refer                   | For index of the                |
 *       | #pmhalPrcmPmicRegulatorId_t)   | gPmhalLP8731Regulator           |
 *       |                                | refer                           |
 *       |                                | #pmhalLP8731RegulatorId_t       |
 *       --------------------------------------------------------------------
 *       This table when translated to code is as below:
 *       pmhalLP8731RegulatorMap_t regulatorMap[
 *          PMHAL_PRCM_PMIC_REGULATOR_COUNT] = {
 *              {
 *                  &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK1],
 *                  I2C_INSTANCE,
 *                  PMIC_I2C_SLAVE_ADDRESS
 *              },
 *              {
 *                  &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK2],
 *                  I2C_INSTANCE,
 *                  PMIC_I2C_SLAVE_ADDRESS
 *              },
 *               ......
 *          };
 *
 * \param   regulatorMap    Pointer to the array of pointers which gives the
 *                          mapping. The array is defined as above.
 *
 * \return  None
 */
void PMHALLP8731ConfigureRegulatorMap(pmhalLP8731RegulatorMapPtr_t regulatorMap);
```

For instance, the custom configuration of PMIC regulator mapping, I2C instance number and PMIC I2C Slave address for the LP8731 could be as shown in Figure 5.

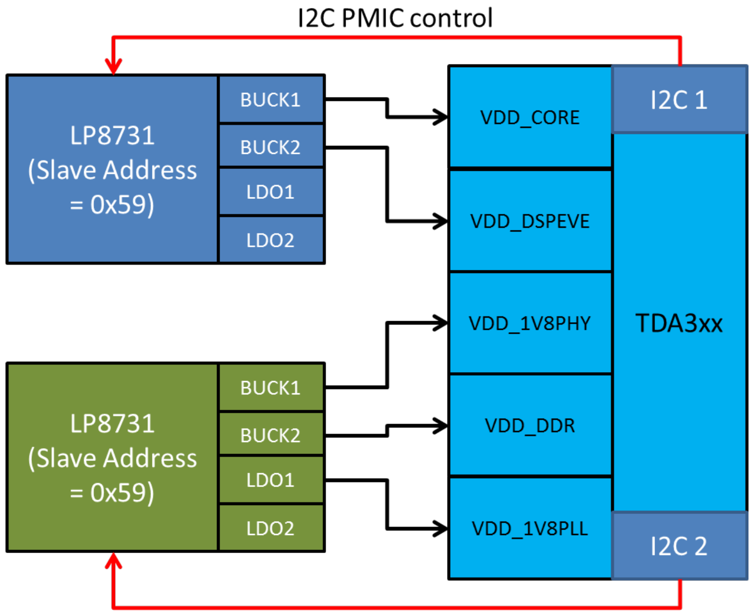


Figure 5. Example Integration of LP8731 PMIC to TDA3xx SoC

In this case the input to the configuration API is as given below. Note that for boards which use the default TI EVM integration, this API need not be called.

```
/* Table mapping from SMPS/LDO to Voltage Rails on the device */
pmhalLP8731RegulatorMap_t         gCustomLP8731RegulatorTable[
    PMHAL_PRCM_PMIC_REGULATOR_COUNT] =
{
    /* HW Regulator for PMHAL_PRCM_PMIC_REGULATOR_CORE */
    {
        &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK1],
        (uint8_t) PMHAL_LP8731_I2C_NUM_1,
        (uint8_t) PMHAL_LP8731_CHIP_ADDRESS
    },
    /* HW Regulator for PMHAL_PRCM_PMIC_REGULATOR_DSPEVE */
    {
        &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK2],
        (uint8_t) PMHAL_LP8731_I2C_NUM_1,
        (uint8_t) PMHAL_LP8731_CHIP_ADDRESS
    },
    /* HW Regulator for PMHAL_PRCM_PMIC_REGULATOR_1V8PHY */
    {
        &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK2],
        (uint8_t) PMHAL_LP8731_I2C_NUM_2,
        (uint8_t) PMHAL_LP8731_CHIP_ADDRESS
    },
    /* HW Regulator for PMHAL_PRCM_PMIC_REGULATOR_DDR */
    {
        &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_BUCK1],
        (uint8_t) PMHAL_LP8731_I2C_NUM_2,
        (uint8_t) PMHAL_LP8731_CHIP_ADDRESS
    },
    /* HW Regulator for PMHAL_PRCM_PMIC_REGULATOR_1V8PLL */
    {
        &gPmhalLP8731Regulator[PMHAL_LP8731_REGULATOR_LDO1],
        (uint8_t) PMHAL_LP8731_I2C_NUM_2,
        (uint8_t) PMHAL_LP8731_CHIP_ADDRESS
    }
```

```
};

PMHALLP8731ConfigureRegulatorMap (gCustomLP8731RegulatorTable);
```

Once this API has been called in the beginning of the application the PMIC driver continues to use the registered mapping for any further communication with the PMIC. Application developers are advised to ensure appropriate I2C Pin Mux configuration and I2C instance power configuration is done before attempting to program the PMIC.

3.3 Setting the Frequency of Modules

Based on the expected CPU operations, the CPUs in the system can be configured to operate at different frequencies as determined by the OPP and speed bin of the device. Setting the OPP of the CPU involves setting the DPLLs to provide the desired clock frequency and setting the PMIC to provide the desired voltage as determined in the AVS EFuse registers. Similarly, different peripherals require different frequencies to meet a given speed of operation (for example, Inter-Integrated Circuit (I2C), Universal Asynchronous Receiver/Transmitter (UART), Multichannel Audio Serial Port (McASP), and so forth).

The generic clocking scheme in the device can be visualized as a tree where a certain input clock to the device is the root node (or Root Clock) of the clock tree and the modules are leaf nodes. Between these two levels are multiple levels consisting of DPLLs, multiplexers and dividers. Configuring a clock frequency for the module requires programming these DPLLs, multiplexers and dividers. The generic view of the device internal clocking is as shown in Figure 6.

The PM software provides two APIs to set and get the frequency of any clock of any given module. The interface of the two APIs is as given below:

```
/**
 * \brief  Set the clock rate of the given module.
 *
 * \param  modId           Module ID
 *                         Refer Enum #pmhalPrcmModuleId_t for values.
 * \param  clkId           Clock Id present in the module
 *                         Refer Enum #pmhalPrcmClockId_t for values.
 * \param  clkRate         new clock rate in Hz to be provided for the
 *                         clockID
 *
 * \retval errorStatus     Status of API call. Can be any of the following,
 *          PM_SUCCESS      Indicates the operation is success
 *          PM_FAIL         Can Indicate the following:
 *                          -modId is not valid.
 *                          -clockRate provided is not supported
 *                          Refer Enum #pmErrCode_t for detailed values.
 */
pmErrCode_t PMLIBClkRateSet(pmhalPrcmModuleId_t modId,
                            pmhalPrcmClockId_t  clkId,
                            uint32_t            clkRate);

/**
 * \brief Get the current clock rate of the given module.
 *
 * \param modId            Module ID
 *                         Refer Enum #pmhalPrcmModuleId_t for values.
 * \param  clkId           Clock Id present in the module
 *                         Refer Enum #pmhalPrcmClockId_t for values.
 * \param  clkRate         new clock rate in Hz returned for the clockID
 *
 * \retval errorStatus     Status of API call. Can be any of the following,
 *          PM_SUCCESS      Indicates the operation is success
 *          PM_FAIL         Can Indicate the following:
 *                          -modId is not valid.
 *                          -clockId provided is not valid
 *                          -Structure provided is not properly initialized
 *                          Refer Enum #pmErrCode_t for detailed values.
 */
pmErrCode_t PMLIBClkRateGet(pmhalPrcmModuleId_t modId,
```

```
pmhalPrcmClockId_t  clkId,
uint32_t            *clkRate);
```

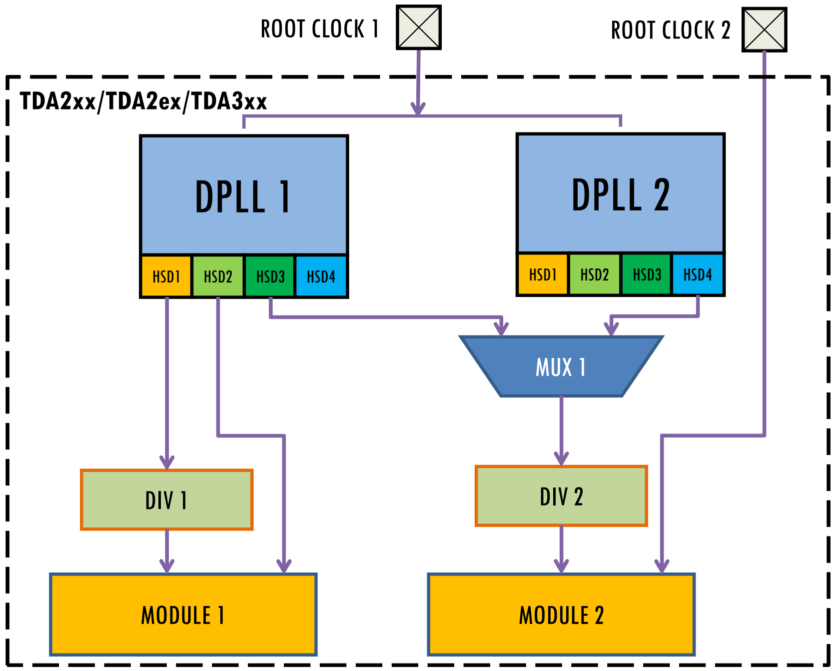


Figure 6. PRCM Representative Clock Tree Structure

The `PMLIBClkRateSet` API configures the necessary DPLL, Mux and divider registers for a given input frequency. The API uses an internal database, which is used to find the necessary DPLL, Mux and Divider programming for a given frequency. The list of supported frequencies for a given clock of a given module can be seen in the following files:

- `<STW Install Directory>\pm\pmlib\tda2xx\pmlib_clk_rate_supported_freq_tda2xx.txt`
- `<STW Install Directory>\pm\pmlib\tda2xx\pmlib_clk_rate_supported_freq_tda2ex.txt`
- `<STW Install Directory>\pm\pmlib\tda3xx\pmlib_clk_rate_supported_freq_tda3xx.txt`

A sample table is shown in Table 3. The first column in the table is the name of the module, the second is the clock name and the third column is the list of supported frequencies in Hz. Each row is a unique combination of module and clock name.

Table 3. Sample Information for Supported Clock Frequencies Captured in the PMLIB Clock Rate Database

Module Name	Clock Name	Supported Frequencies (Hz)
PMHAL_PRCM_MOD_ATL	PMHAL_PRCM_CLK_ATL_GFCLK	266000000, 32786, 451584000
PMHAL_PRCM_MOD_ATL	PMHAL_PRCM_CLK_ATL_L3_GICLK	266000000
PMHAL_PRCM_MOD_IO_SRCOMP_CORE	PMHAL_PRCM_CLK_COREAON_IO_SRCOMP_GFCLK	20000000
PMHAL_PRCM_MOD_EVE1	PMHAL_PRCM_CLK_EVE1_GFCLK	535000000, 650000000
PMHAL_PRCM_MOD_EVE2	PMHAL_PRCM_CLK_EVE2_GFCLK	535000000, 650000000
PMHAL_PRCM_MOD_EVE3	PMHAL_PRCM_CLK_EVE3_GFCLK	535000000, 650000000
PMHAL_PRCM_MOD_EVE4	PMHAL_PRCM_CLK_EVE4_GFCLK	535000000, 650000000
PMHAL_PRCM_MOD_UART6	PMHAL_PRCM_CLK_UART6_GFCLK	192000000, 48000000
PMHAL_PRCM_MOD_IPU1	PMHAL_PRCM_CLK_IPU1_GFCLK	212800000
PMHAL_PRCM_MOD_IPU2	PMHAL_PRCM_CLK_IPU2_GFCLK	212800000
PMHAL_PRCM_MOD_IVA	PMHAL_PRCM_CLK_IVA_GCLK	388333333, 430000000, 532000000
PMHAL_PRCM_MOD_SL2	PMHAL_PRCM_CLK_IVA_GCLK	388333333, 430000000, 532000000

Table 3. Sample Information for Supported Clock Frequencies Captured in the PMLIB Clock Rate Database (continued)

Module Name	Clock Name	Supported Frequencies (Hz)
PMHAL_PRCM_MOD_TIMER2	PMHAL_PRCM_CLK_L4PER_L3_GICLK	266000000
PMHAL_PRCM_MOD_TIMER2	PMHAL_PRCM_CLK_L4PER_L4_GICLK	133000000
PMHAL_PRCM_MOD_TIMER2	PMHAL_PRCM_CLK_TIMER2_GFCLK	20000000, 10000000, 32786, 22579200, 451584000
PMHAL_PRCM_MOD_TIMER3	PMHAL_PRCM_CLK_L4PER_L3_GICLK	266000000
PMHAL_PRCM_MOD_TIMER3	PMHAL_PRCM_CLK_L4PER_L4_GICLK	133000000
PMHAL_PRCM_MOD_TIMER3	PMHAL_PRCM_CLK_TIMER3_GFCLK	20000000, 10000000, 32786, 22579200, 451584000

NOTE: The file shown in Table 3 is an auto generated file and should not be modified manually. In order to add any frequency to this table (in case the frequency you are looking for is not available), see Section 3.3.1. TI provides the sample database corresponding to 20 MHz input SYSCLK1 frequency as a part of the STW software package. To include support for any other SYSCLK1 frequencies kindly contact your TI representative.

The API makes sure when the frequency of a CPU is changed such that the OPP shifts, the PMIC is also programmed to change the voltage to correspond to this OPP shift. The API also takes into account the ganging of voltage rails which can happen outside the device boundaries. The user input table in the format as shown in Table 5. Table 5 is parsed through to find all the dependent CPUs in the voltage rails, which are ganged and if the new voltage is found to satisfy the OPP requirement of any ganged CPU the voltage is changed along with the DPLL configuration.

NOTE: The Clock Rate APIs do not take care of dependencies between the different paths in the database. For instance, if the same DPLL is sourcing multiple clocks and two different clocks require frequencies such that the DPLL needs to be locked to two different frequencies, the API/Database does not recognize this conflict and would program the DPLL to the latest configuration to which it has been asked to be programmed. The Application developer needs to be aware of any such conflicts to avoid unwanted behavior from the device.

The `PMLIBClkRateSet` and `PMLIBClkRateGet` APIs need an initialization step that populates the root clock frequencies and the information regarding the ganging of voltage rails in order for the APIs to work correctly. The two initialization structures are shown in Table 4 and Table 5. The root clock frequencies are represented as an array of 32-bit unsigned numbers. The list of root clock frequencies that need to be populated for TDA2xx/TDA2ex and TDA3xx devices is given in Table 4.

Table 4. TDA2xx/TDA2ex/TDA3xx Root Clock List

TDA2xx/TDA2ex Root Clock List	TDA3xx Root Clock List
PMHAL_PRCM_ROOT_CLK_PCIESREF_ACS	PMHAL_PRCM_ROOT_CLK_SYS_CLKIN1
PMHAL_PRCM_ROOT_CLK_RMII	PMHAL_PRCM_ROOT_CLK_SYS_CLKIN2
PMHAL_PRCM_ROOT_CLK_SYS_32K	
PMHAL_PRCM_ROOT_CLK_SYS_CLKIN1	
PMHAL_PRCM_ROOT_CLK_SYS_CLKIN2	

The ganging of voltage rails is represented as an array of structures for each voltage rail in the device. For example, if the voltage rails VD_DSPEVE, VD_IVA and VD_GPU are ganged together outside the device on the board one can populate the voltage ganging input structure as shown in Table 5.

Table 5. Example Structure for Ganging of Voltage Rails

Voltage ID	List of Ganged Voltage Rails	Number of Ganged Rails
PMHAL_PRCM_VD_MPU	NA	0
PMHAL_PRCM_VD_CORE	NA	0
PMHAL_PRCM_VD_IVAHD	PMHAL_PRCM_VD_DSPEVE, PMHAL_PRCM_VD_GPU	2
PMHAL_PRCM_VD_DSPEVE	PMHAL_PRCM_VD_IVAHD, PMHAL_PRCM_VD_GPU	2
PMHAL_PRCM_VD_GPU	PMHAL_PRCM_VD_IVAHD, PMHAL_PRCM_VD_DSPEVE	2

TI provides reference software for both root clock list and ganging of voltage rails corresponding to TI EVMs in the following files:

- `<STW Install Directory>\starterware_\pm\pmlib\tda2xx\pmlib_boardconfig.c`
- `<STW Install Directory>\starterware_\pm\pmlib\tda3xx\pmlib_boardconfig.c.`

Table 6 gives the average time the `PMLIBClkRateSet` and `PMLIBClkRateGet` APIs take to set and get the frequency of a given clock. The measurements were done on A15 (in TDA2xx/TDA2ex) running at 750 MHz and M4 (in TDA3xx) running at 212.8 MHz.

Table 6. Execution time for PMLIB Clock Rate APIs

Application Interface	A15 Time Taken (µs)		M4 Time Taken (µs)	
	MAX	AVG	MAX	AVG
PMLIBClkRateSet	2461.65	716.24	690.6	274.2
PMLIBClkRateGet	1375.1	387.5	519.6	236.4

3.3.1 Modifying the Clock Frequency Database

The clock frequency database is generated using an excel sheet that captures the DPLL configuration, multiplexer and divider configurations for each module and its corresponding clock in the device. The excel sheets are located in the following folders:

- `<STW Install Dir>\pm\pmlib\tda2xx`
- `<STW Install Dir>\pm\pmlib\tda3xx`

The flow to convert the excel sheet to code is as shown in Figure 7. The different columns of the excel sheet are also described in Figure 7.

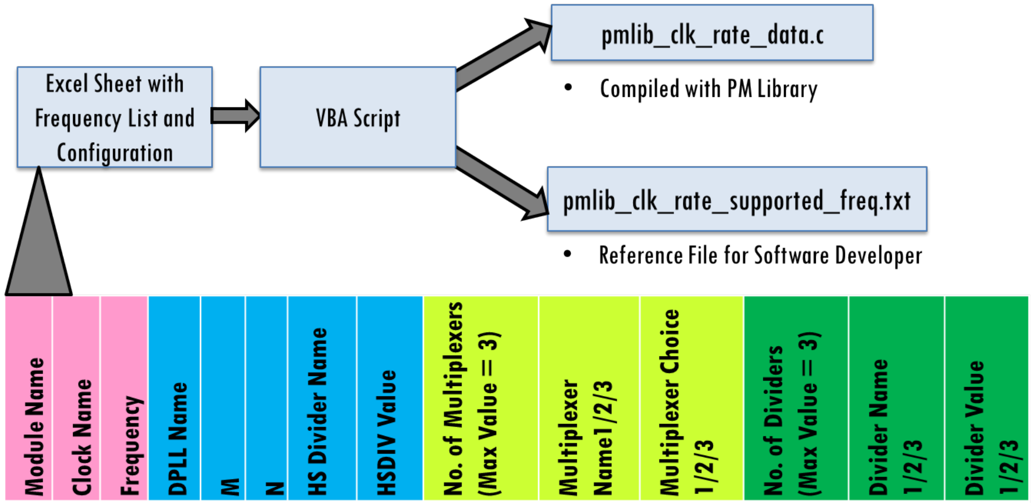
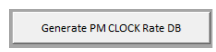


Figure 7. PMLIB Clock Rate Database Excel Sheet Generator Flow

In order to add a new frequency in the database, do the following steps:

1. In the corresponding device clock tree excel sheet, find the module and clock pair whose frequency needs to be added.
2. For the new clock frequency, insert a row after the current list of frequencies.
3. Populate this new row, with the module name, clock name, new frequency, DPLL, multiplexer and divider configurations.
4. Once done with the editing, click on the Generate PM Clock Rate DB  button (refer to Cell B1 in the excel sheet). Two files (C file and text file) are generated (overwriting the existing default C and text file) in the same directory as the excel sheet.
5. Confirm that the text file shows the updated list of supported frequencies.
6. Re-build the PMLIB library with the generated C file.

4 System Power Initialization

In order to make sure the device power consumption is optimal, the software should ensure that only the modules required by the use case are kept ON and all the other modules are placed in OFF state.

Setting the module to have different power states involves programming the Power Domain (Level 3), the Clock Domain (Level 2) and Module (Level 1) to appropriate values that allow the modules to be in different power states. Essentially the module power states can be abstracted to fall in three categories:

- DISABLED – Lowest Power Configuration
  - Power Domain State: OFF or RETENTION
  - Clock Domain State: SW_SLEEP or HW_AUTO
  - Module State: DISABLED or HW_AUTO
- AUTO CLOCK GATE (AUTO_CG) – Clocks disabled when module is not used or else clocks are on
  - Power Domain State: ON when module is active, ON or RETENTION or ON_INACTIVE when module is not used.
  - Clock Domain State: HW_AUTO (Clocks are running when the module is used and the clocks are gated when the module is not used.)
  - Module State: HW_AUTO

- ALWAYS ENABLED – Highest Power Configuration
  - Power Domain State: ON
  - Clock Domain State: SW_WKUP (Clocks are always on)
  - Module State: ENABLED

Different modules have different default power states after the device boots up. Modules like MMC2, MLB_SS, SATA, OCP2SCP1, OCP2SCP3, USB_OTG_SS1, USB_OTG_SS2, USB_OTG_SS3, USB_OTG_SS4, PCIESS1, PCIESS2, and so forth are disabled by default. Modules like EVE1/2/3/4 have their power domain ON (thus drawing leakage power) but have their clocks gated. ROM also initializes certain peripherals to ON state to be able to boot from them.

Given this varied state of power of different modules at boot, the Secondary Boot Loader (SBL) should initialize the system intelligently to make sure only the modules getting used for a given use case are ENABLED and the ones not getting used are DISABLED.

Suggested module power states that can be applied when deciding the optimal power state of the TDA2xx, TDA2ex and TDA3xx devices are summarized in Table 7.

Table 7. Suggested Module Power States for TDA2xx/TDA2ex/TDA3xx Modules

Module Name	Power State after ROM boot	SBL Desired Action
MPU Core 0 and Core 1 (1)	ON	Force Off Core 1 when not used
IPU, DSP1 and 2	OFF (2)	Initialize core when valid application image is present. Power Off if not.
EVE1/2/3/4 (3)	ON (Clock Gated)	Initialize core when valid application image is present. Power Off if not.
Peripherals	Varied	Disable Module if not used.

(1) Valid only for TDA2xx.
(2) IPU, DSP1 and DSP2 are ON after ROM boot in TDA3xx.
(3) EVE IP not present in TDA2ex.

The STW PMLIB provides the necessary APIs to be able to set the power state of the module and also get the power status of a given module. The two APIs are as below:

```
/**
 * \brief   This API configures the given module to the desired power state.
 *
 * \param   inputTable      Table of modules and their desired power and
 *                          clock state.
 * \param   numConfig       Number of entries in the system configuration table.
 * \param   timeout         Desired time out for which one should wait for each
 *                          of the modules to reach its desired power state.
 *                          PM_TIMEOUT_NOWAIT (0) - The API does not wait.
 *                          PM_TIMEOUT_INFINITE (0xFFFFFFFF) - The API waits
 *                          infinitely.
 *                          Any other value - The API waits to any one of the
 *                          events to happen first : (1) Success of operation
 *                          (2) The timeout is reached.
 * \param   resultReturn    Table which returns success or error codes got
 *                          while programming the power state. Useful for debug
 *                          One must allocate the same number of entries as the
 *                          input table to ensure the API has sufficient space
 *                          to return the error or pass codes. The structure
 *                          returns in the format
 *                          [module, return code (success/fail code)] for all
 *                          the modules given in a one to one mapping.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the desired power state was met.
 *                  PM_FAIL     If the desired power state was not met. One must
 *                              check the resultReturn to check the cause for
```

```
 *                               failure.
 */
pmErrCode_t PMLIBSysConfigSetPowerState(
    const pmlibSysConfigPowerStateParams_t *inputTable,
    uint32_t                                numConfig,
    uint32_t                                timeout,
    pmlibSysConfigErrReturn_t              *resultReturn);

/**
 * \brief   This API is used to get the power state for a given module.
 *
 * \param   moduleId        Module ID of the module one is interested in. Refer
 *                          #pmhalPrcmModuleId_t for details.
 * \param   currState       Returns the final state of the module. Refer
 *                          #pmlibSysConfigPowerState_t for details.
 * \param   detailedState   This is an optional parameter which can be used to
 *                          return the detailed state of the module broken
 *                          down into module state, clock state and power state.
 *                          If one is not interested in knowing the detailed
 *                          state one can put NULL for this parameter.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the status is obtained correctly.
 *                  PM_BADARGS  If currState pointer is NULL.
 */
pmErrCode_t PMLIBSysConfigGetPowerState(
    pmhalPrcmModuleId_t            moduleId,
    pmlibSysConfigPowerState_t    *currState,
    pmlibSysConfigDetailedState_t *detailedState);
```

The *PMLIBSysConfigSetPowerState* API configures the power domain, clock domain and the module registers at the PRCM level and additionally the module SYSCONFIG registers wherever available to be able to place the module in any one of the three power states *DISABLED, AUTO_CG or ALWAYS ENABLED*. The API also configures the static dependencies for the clock domain in which the module resides.

The API takes a table of input modules and their desired power states to initialize the power state of the system. An example input table is as shown in Table 8.

Table 8. Example Input Table for PMLIB Set System Configuration

Module Name	Power State
PMHAL_PRCM_MOD_DSP1	PMLIB_SYS_CONFIG_ALWAYS_ENABLED
PMHAL_PRCM_MOD_MCASP1	PMLIB_SYS_CONFIG_DISABLED
PMHAL_PRCM_MOD_IPU1	PMLIB_SYS_CONFIG_AUTO_CG

For subsystems that have a reset configuration available with them, the API also performs the subsystem reset management. For example, when the EVE1 subsystem is configured to be ALWAYS ENABLED, the API enables the EVE subsystem power, clock and module and lifts the subsystem reset (RM_EVE1_RSTCTRL.RST_EVE1 = 0). Similarly, when the EVE1 subsystem is configured to be DISABLED, the subsystem reset is asserted by the API. The API does not configure any reset for a module configured for AUTO_CG.

NOTE: The PMLIBSysConfigSetPowerState does not take care of dependencies between modules. For instance, if a module A requires module B and C to be enabled before module A can be enabled, the application must call the enable operation of B and C before enabling A.

An example sequence of the *PMLIBSysConfigSetPowerState* API to initialize the DSP1 and IPU1 subsystem is given below:

```
#include "pmlib_sysconfig.h"

pmlibSysConfigPowerStateParams_t inputTable[] =
{{PMHAL_PRCM_MOD_DSP1,  PMLIB_SYS_CONFIG_ALWAYS_ENABLED},
 {PMHAL_PRCM_MOD_IPU1,  PMLIB_SYS_CONFIG_ALWAYS_ENABLED}};

const uint32_t numTableEntries = sizeof (inputTable) /
                                 sizeof (pmlibSysConfigPowerStateParams_t);

pmErrCode_t               status = PM_SUCCESS;
/* Initialize the system modules to be enabled. Subsystem resets if any are
 * de-asserted */
status = PMLIBSysConfigSetPowerState(inputTable, (uint32_t) numTableEntries,
                                         PM_TIMEOUT_INFINITE,
                                         NULL);
if (PM_SUCCESS == status)
{
    /* Load the code for the DSP 1 subsystem */
    DSP1_LoadCode();
    /* Configure AMMU for IPU1 */
    IPU1_ConfigAmmu();
    /* Load the IPU1 Code*/
    IPU1_LoadCode();
    /* Lift the IPU1 C0 CPU reset to start the CPU execute instructions */
    PMHALResetRelease(PMHAL_PRCM_RG_IPU1_CPU0_RST, PM_TIMEOUT_INFINITE);
    /* Lift the DSP CPU reset to start the CPU execute instructions */
    PMHALResetRelease(PMHAL_PRCM_RG_DSP1_RST, PM_TIMEOUT_INFINITE);
```

Similarly, the PMLIBSysConfigGetPowerState API can be used to know the power status of the module. The API returns the derived power state (DISABLED, AUTO_CG or ALWAYS ENABLED) and the individual power domain, clock domain and module status for detailed analysis.

Table 9 gives the average time the API takes for the API to set and get the power state per module. The measurements were done on A15 (in TDA2xx/TDA2ex) running at 750 MHz and M4 (in TDA3xx) running at 212.8 MHz.

Table 9. Execution time for PMLIB System Configuration APIs

Application Interface	Power State	A15 Time Taken (µs)	M4 Time Taken (µs)
PMLIBSysConfigSetPowerState	ALWAYS_ENABLED	19.5	142.7
	DISABLED	51.4	258.8
	AUTO_CG	26.4	190.4
PMLIBSysConfigGetPowerState	NA	10.7	68.5

4.1 Knowing System Power State With GEL Script

In order to know the state of the modules in the system at use case development time, the developer can use Code Composer Studio™ GEL scripts `TDA2xx_PRCM_Get_Config.gel`, `TDA3xx_PRCM_Get_Config.gel` or `TDA2ex_PRCM_Get_Config.gel` to find the power state of all the modules in the device. Once the device is booted, connect to the device master from CCS and run the `PRCM_GetConfig` GEL menu as shown in Figure 8. This feature is available in GEL version 9 onwards.

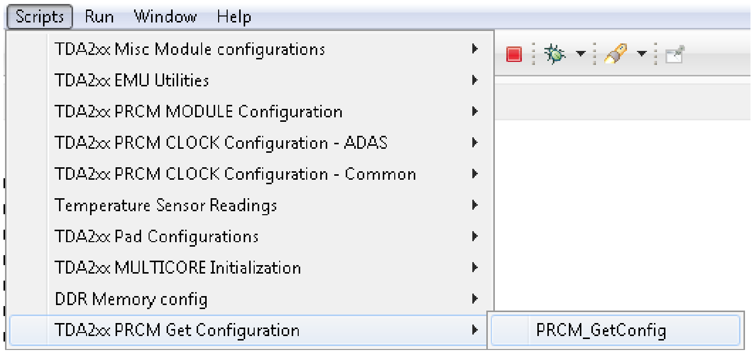


Figure 8. GEL Script Menu Option Screen Shot

The GEL function prints the PRCM state of all the modules in the system. Based on their ON/OFF state, the use case developer can appropriately call the *PMLIBSysConfigSetPowerState* APIs to enable only those modules that are required and disable the rest. The output format of the GEL script is as shown below:

```
GEL Output: ==========================================
GEL Output: Module : <MODULE NAME> (CD_<CLOCK_DOMAIN>, PD_<POWER_DOMAIN>)
GEL Output:          Module State : <MODULE_STATE_FROM_CLKCTRL>
GEL Output:          Clock State : <CLOCK_STATE_FROM_CLKSTCTRL>
GEL Output:          Power State : <POWER_STATE_FROM_PWRSTCTRL>
GEL Output:          Final State : <Final state as derived from Module,
                                              Clock and Power State>
GEL Output: ==========================================
```

5 Dynamic CPU Power Management

Dynamic power management of the CPUs involves setting the CPU power state to a lower power state when the CPU has nothing to do and then wakes up when the CPU has to resume its operations. The process of dynamic CPU power management is as shown in Figure 9. The CPU context is maintained every time the CPU goes to low-power mode. Interrupts that are configured during the normal CPU operation can be configured as wake up events, which bring the CPU out of its low-power state.

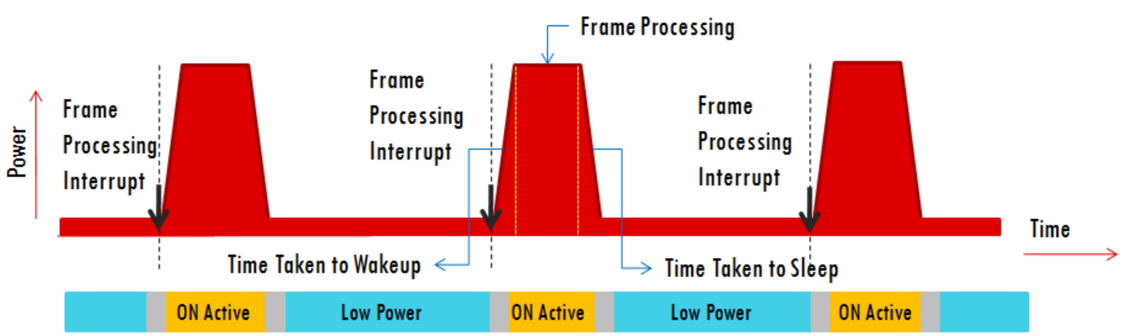


Figure 9. Dynamic CPU Power Management

The following subsections provide an in-depth analysis of the different power states of the CPUs in the system (MPU, DSP, IPU, EVE) and the time taken for going into and waking up from the low-power states. Based on the latency requirements, the application developer can decide on a fixed-power state for the CPUs for the low-power state. The different power states for the different CPUs in the system, from highest (left most) to lowest (right most), are summarized in Table 10. The following sections dive deep into the power and clock state of the different subsystems and the programming sequence to achieve the desired power state.

Table 10. Different Power States supported by CPU Subsystems

CPU Subsystem	Highest Power State				Lowest Power State
MPU	On	Core 1 Forced Off	Core 1 Forced Off, Core 0 Idle	Subsystem Auto Clock Gate	Subsystem Retention
DSP	On	CPU Idle	Subsystem Standby	Subsystem Auto Clock Gate	Subsystem Off
IPU	On	CPU Idle	Subsystem Standby	Subsystem Auto Clock Gate	Subsystem Off
EVE	On	CPU Idle	Subsystem Standby	Subsystem Auto Clock Gate	Subsystem Off

Some recommended power states for the different CPUs are shown below:

- MPU : Subsystem Retention
  `<STW_Install_Dir>\examples\pm\cpuidle\main_a15host.c`
- IPU : Subsystem Auto Clock Gate
  `<STW_Install_Dir>\examples\pm\cpuidle\main_m4.c`
- DSP : Subsystem Auto Clock Gate
  `<STW_Install_Dir>\examples\pm\cpuidle\main_c66x.c`
- EVE : Subsystem Auto Clock Gate
  `<STW_Install_Dir>\examples\pm\arp32_cpuidle\main_arp32.c`

5.1 MPU Power Management

NOTE: This section is valid for TDA2xx. TDA2ex has a single A15 core in the MPU subsystem.

5.1.1 MPU PM Blocks

The MPU (Dual Cortex®-A15 Subsystem) has its internal Local PRCM (MPU_PRCM) and is also controlled by the system-on-chip (SoC) level Global PRCM. The MPU_PRCM is responsible for power management of the Cortex-A15 CPU power domains and corresponding L1 Cache highlighted in sea green and light green boxes in Figure 10.

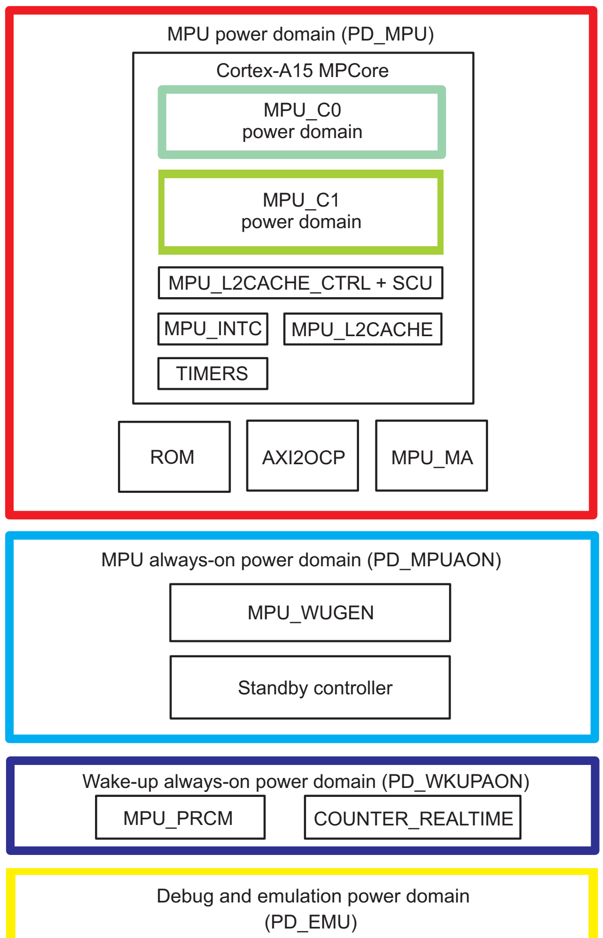


Figure 10. MPU Subsystem Power Domains

The global PRCM is responsible for the MPU Power domain including the L2 Cache, Interrupt Controllers, and MPU Memory Adapter, and so forth (highlighted in red in Figure 10).

The MPU Wake up Generator (WUGEN) responsible for waking up MPU from a low-power state when a wake up event occurs is in the MPU always on power domain along with the MPU standby controller.

The MPU MPU_PRCM resides in the always on wake up power domain (PD_WKUPAON).

In addition to the standard power-management technique supported in the device, the MPU subsystem also employs a SR3-APG (SmartReflex3 automatic power gating) power-management technology to reduce leakage.

This technology allows for full logic and memory retention on MPU_C0 and MPU_C1, and is controlled by the MPU_PRCM. The SR3-APG power management can be enabled by setting the PRM_PSCON_COUNT [24] HG_EN bit.

5.1.2 MPU PM States

This section discusses the different power states of the MPU subsystem and understands the power down and wake up latencies based on the power savings. Contact your TI representative to get access to the TDA2xx/TDA2ex Power Estimation Spread Sheet and analyze the exact power savings for the different supported low power modes.

- ***CASE 1: MPU ON (Core 0 (C0) and Core 1 (C1) On)***

  In this configuration, the Cortex-A15 CPUs are both alive and running their respective software. This is the highest power consumption configuration and the power consumption is determined by the kind of operations the A15 is performing. The sub-modules in the MPU subsystem are ON and the clocks are enabled for the subsystem and the CPUs.

- ***CASE 2: MPU C1 Forced Off***

  In this configuration, the MPU C1 is forced off and the MPU C0 is alive and running its own software. In this configuration, the MPU C1 logic, L1 Cache is off, the clocks to the MPU C1 are gated and the MPU C1 LPRM shows the CPU to be in power off mode.

  In order to force the CPU1 to OFF state one must perform the following operations:

  1. Clear the SCTLR.C bit or HSCTLR.C bit if in Hypervisor mode, to prevent additional data cache allocation.
  2. Clean and invalidate all data from the L1 data cache. This prevents any new data cache snoops or data cache maintenance operations from other processors in the multiprocessor being issued to this processor.
  3. Switch the processor from Symmetric Multiprocessing (SMP) mode to Asymmetric Multiprocessing (AMP) mode by clearing the ACTLR SMP bit. Clearing the SMP bit enables the processor to be taken out of coherency by preventing the processor from receiving cache, TLB, or BTB maintenance operations broadcast by other processors in the multiprocessor.
  4. Ensure that the system does not send interrupts to the processor that is being powered down.
  5. Execute an ISB instruction to ensure that all of the CP15 register changes from the previous steps have been committed.
  6. Execute a DSB instruction to ensure that all cache, TLB and branch predictor maintenance operations issued by any processor in the multiprocessor before the SMP bit was cleared have completed.
  7. Execute a WFI instruction and wait until the STANDBYWFI output is asserted to indicate that the processor is in idle and low-power state.

  The MPU CPU 1 can be forced off using the STW PM API as given below:

```
/**
 * \brief   Enter CPU1 into FORCE_OFF mode.
 *
 *          This function can be used by the application to enter CPU1
 *          into FORCE_OFF mode when no binary is loaded for CPU1.
 *
 * \param   none
 *
 * \return  none
 */
void PMLIBCpu1ForcedOff(void);
```

  An example programming sequence to put the MPU C1 to forced off is as given below:

```
MPU_WUGEN_1_DisableAll();
/* Flushing the DCache is required to ensure Core 0 does not get
 * pipeline stalled when the cache is enabled later and the
 * cache invalidate is performed. */
CP15DCacheCleanFlush();
PMLIBCpu1ForcedOff();
```

The time taken to bring the CPU1 to forced off state when MPU is operating at 750 MHz and a GP Timer operating at 20 MHz is used to measure the time taken is given in Table 11.

Table 11. MPU CPU1 Forced Off Time Profile

Function	Time Taken (micro second (µs))
MPU_WUGEN_1_DisableAll();	1.65
CP15DCacheCleanFlush();	7668.6
PMLIBCpu1ForcedOff();	7.5

Putting MPU C1 to a forced off state is a one-time configuration and is typically done in the SBL when there is no application that is to run on CPU1.

CAUTION

In order to wake up the CPU1 from a forced off state the TDA2x would require a full system reboot.

- ***CASE 3: MPU C1 Forced Off and C0 in IDLE***

  In this configuration, the MPU C1 is put to a forced off state with the steps as mentioned in Case 2. Additionally the CORE 0 is made to go to IDLE state whenever the CORE 0 has nothing to run and is again woken up by an event when the CORE 0 has to resume processing.

  The steps to take the CORE 0 to IDLE are shown below:

  1. Program the MPU LPRM Domain state to the desired power state.
  2. Execute WFI instruction.

  The STW API to enable the CORE 0 going into low-power mode is shown below:

```
/**
 * \brief   Enter the CPU specified into the given low power state.
 *
 * NOTE:    To keep the ADAS power management simple, we only
 *          support one low-power mode per CPU. The power state is
 *          used in the API to enable supporting multiple low power
 *          states in future w/o breaking compatibility.
 *
 * \param   pwrst   Low power state to enter into. Refer enum
 *                  #pmhalPrcmPdState_t for details.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the desired power state was met.
 *                  PM_FAIL     If the desired power state was not met.
 */
pmErrCode_t PMLIBCpuIdle(pmhalPrcmPdState_t pwrst);
```

  The possible inputs to this API are shown below. Program the global PRCM along with the MPU_PRCM to be able to get to a lower MPU Power state. An example of this is shown in CASE 4.

```
PMHAL_PRCM_PD_STATE_OFF = 0U,
/**< Power to the domain is off. */
PMHAL_PRCM_PD_STATE_RETENTION = 1U,
/**< Power to the domain is in retention mode and it will not be functional
 *   but the memory will be retained. */
PMHAL_PRCM_PD_STATE_ON_INACTIVE = 2U,
/**< Power to the domain is ON INACTIVE. */
PMHAL_PRCM_PD_STATE_ON_ACTIVE = 3U,
/**< Power to the domain is ON. */
```

  The usage of this API (shown below) when CPU 0 wants to go to an IDLE state with no power transition:

```
/* Set the CORE 0 to IDLE state with no Power Domain Transition */
status = PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_ON_ACTIVE);
```

In order to wake up the CORE 0 based on a certain interrupt from low power state one must program the MPU WUGEN (Wake Up GENerator) module. The WUGEN unit is responsible for generating a wake up event from the incoming interrupts and enable bits. The WUGEN is implemented in MPU Always-On power domain.

MPU_WUGEN has a 160-bit enable fields (from WKG_ENB_A_x to WKG_ENB_E_x for MPU_Cx, where x = 0 or 1), which define the interrupt that wakes up the MPU cores. Note that each MPU core cannot be independently woken up by interrupts. Instead, an enabled interrupt wakes up both MPU cores (except if MPU_C1 is in FORCED_OFF state). Therefore, the MPU_WUGEN is designed to handle interrupts for both MPU cores and generates a single wake-up request.

All interrupts are enabled after reset as wake up events. Software must first disable the WUGEN events during the interrupt initialization and then set the wake up event enable bit that corresponds to the desired interrupt. A given interrupt for a given MPU core is either enabled at both MPU_INTC and MPU_WUGEN, or disabled at both; no other combination is allowed.

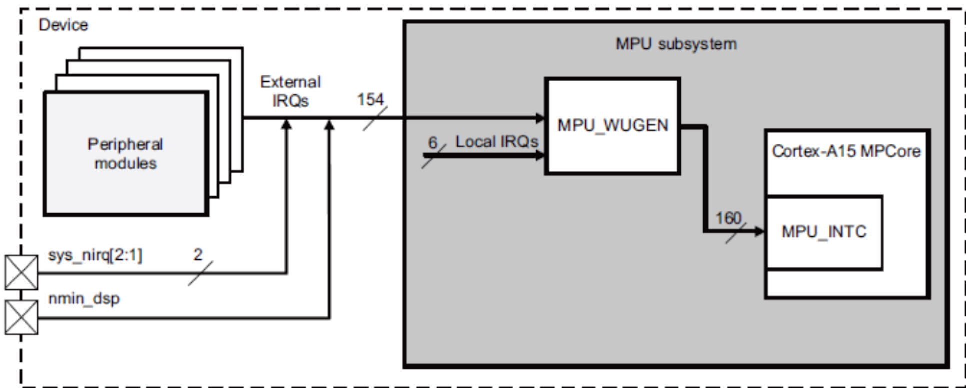


Figure 11. MPU Wake Up Generator Block in MPU Subsystem

The following STW APIs can be used to initialize and enable the MPU WUGEN to enable the wake up event:

```
/**
 * \brief   This API is used to initialize MPU_WUGEN. All wake up events will
 *          be disabled after initialization.
 *
 * \param   None
 *
 * \return  None.
 *
 **/
void MPU_WUGEN_Init(void);

/**
 * \brief   This API enables the CORE 0 wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
void MPU_WUGEN_0_Enable(uint16_t intrNum);

/**
 * \brief   This API enables the CPU1 wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
```

```
void MPU_WUGEN_1_Enable(uint16_t intrNum);
```

The latency to enter the CPU IDLE state and wake up from it is given in Table 12.

- ***CASE 4: MPU Subsystem Auto Clock Gate (C1 Forced Off and C0 in Clock Gate)***

  In this configuration the MPU C1 is put to a forced off state with the steps mentioned in Case 2. Additionally, the CORE 0 along with the MPU Clock Domain is made to go to the Auto Clock Gate State whenever the CORE 0 has nothing to run and is again woken up by an event when the CORE 0 has to resume processing.

  The steps to take the CORE 0 and subsequently CD_MPU to clock gate state are shown below:

  1. Program the MPU Clock Domain state to HW_AUTO Mode.
  2. Program the MPU LPRM Domain state to retention state.
  3. Execute WFI instruction.

```
/* Set MPU_PD to On Active */
PMHALPdmSetPDState(PMHAL_PRCM_PD_MPU, PMHAL_PRCM_PD_STATE_ON_ACTIVE,
                           PM_TIMEOUT_NOWAIT);
/* Set MPU_CD to HW_AUTO */
PMHALCMSetCdClockMode((pmhalPrcmCdId_t) PMHAL_PRCM_CD_MPU,
                              PMHAL_PRCM_CD_CLKTRNMODES_HW_AUTO,
                              PM_TIMEOUT_NOWAIT);
/* Put Core 1 to forced off */
PMLIBCpu1ForcedOff();
/* Program MPU LPRM and execute WFI */
status = PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);
```

- ***CASE 5: MPU Subsystem Retention (C1 Forced Off and C0 in retention)***

  In this configuration, the MPU C1 is put to a forced off state with the steps as mentioned in Case 2. Additionally, the CORE 0 along with the MPU Power Domain is made to go to Closed Switch Retention State whenever the CORE 0 has nothing to run and is again woken up by an event when the CORE 0 has to resume processing.

  The RETENTION low-power state is not natively supported by the MPU_CLUSTER. This mode is implemented with SR3-APG power-management technology. The MPU subsystem power-management hardware is designed to ensure that the system does not have an L1 cache coherency problem when putting both MPU cores in retention mode. In this mode, the MPU_Cx logic is in full retention with all memory content preserved by keeping the array of memories fully powered and the logic of the memory peripheries shut down. In slow wake-up mode, memories are put into retention to prevent more leakage.

  The steps to take the CORE 0 and subsequently PD_MPU to retention are shown below:

  1. Program the Retention mode for MPU Subsystem.
  2. Program the MPU Power Domain state to RET mode.
  3. Program the MPU Clock Domain state to HW_AUTO Mode.
  4. Program the MPU CPU Domain state to RET mode.
  5. Execute WFI instruction.

  The STW APIs to configure the Retention and Fast Ramp-up are shown below:

```
/**
 * \brief   Function to set the HG_EN bit for MPU Mercury retention.
 * \param   None.
 * \return  None.
 */
void PMHALMpuLprmSetMercuryRetention(void);

/**
 * \brief   Function to set the Mercury Retention ramp parameters. This is to
 *          be used only If the HG_EN bit is set.
 *
 * \param   hgRampParam     Structure which holds the parameters to be set for
 *                          correct mercury retention ramp.
 *                          If fast ramp is used then no need to fill the
 *                          hgSlowRampTime field.
```

```
 * \return  status          PM_SUCCESS If the ramp parameters are set
 *                          correctly.
 *                          PM_BADARGS If the slow ramp is enabled and the
 *                                     ramp time provided is 0.
 */
int32_t PMHALMpuLprmSetHgRampParams(
const pmhalMpuLprmHgRampParams_t *hgRampParam);
```

The usage of these APIs is shown below when CPU 0 wants to go to Retention state along with the power domain configuration to retention:

```
/* Enable Fast Ramp Up */
pmhalMpuLprmHgRampParams_t hgRampParam = {1, 0};
/* Program the MPU to AUTO_CG. This will put the MPU PD to Retention */
pmlibSysConfigPowerStateParams_t inputTable = {
PMHAL_PRCM_MOD_MPU, PMLIB_SYS_CONFIG_AUTO_CG};

/* CONFIG MPU MPU_PRCM: Enable FastRamp-up in Retention */
PMHALMpuLprmSetHgRampParams(&hgRampParam);
/* CONFIG MPU MPU_PRCM: Enable Mercury Retention */
PMHALMpuLprmSetMercuryRetention();

/* CONFIG GLOBAL PRCM: Request AUTO CG and Retention */
status = PMLIBSysConfigSetPowerState(
&inputTable, 1, PM_TIMEOUT_NOWAIT, NULL);

/* Put MPU LPRM to retention and execute WFI */
status = PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);
```

In order to wake up the MPU from retention, program the MPU WUGEN as described in CASE 3.

Table 12 summarizes the MPU power and clock states for the different cases seen and the associated latency to bring the MPU to low power state and wake up.

For an example implementation, see the STW examples in the folder: `<STW Install Dir>\examples\pm\cpuidle`.

Table 12. MPU Power Management States and Latency

		CASE 1		CASE 2		CASE 3		CASE 4		CASE 5	
		MPU C0	**MPU C1**	**MPU C0**	**MPU C1**	**MPU C0**	**MPU C1**	**MPU C0**	**MPU C1**	**MPU C0**	**MPU C1**
Power state		Always enabled	Always enabled	Always enabled	Disabled	Always enabled	Disabled	Auto	Disabled	Auto	Disabled
Processing	Reference profile	Dhrystone/Max/Stall/ memcpy	Dhrystone/Max/Stall/ memcpy	Dhrystone/Max/Stall/ memcpy	idle	idle	idle	idle	idle	idle	idle
	Utilization (% Active)	100	100	100	0	0	0	0	0	0	0
HGEN (LPRM Mercury Retention Enable)		any		any		any		1		1	
HG_RAMPUP (LPRM Mercury Retention Fast Ramp Enable)		any		any		any		1		1	
MPU LPRM programed Values	LPRM Power State	any	any	any	FORCED_OFF	ON	FORCED_OFF	RET	FORCED_OFF	RET	FORCED_OFF
	LPRM ClkTrCtrl	any	any	any	HW_ AUTO	HW_ AUTO	HW_ AUTO	HW_ AUTO	HW_ AUTO	HW_ AUTO	HW_ AUTO
MPU Core State MPU Programmed Values	Logic	ON	ON	ON	OFF	ON	OFF	SR3-APG	OFF	SR3-APG	OFF
	L1$	ON	ON	ON	OFF	ON	OFF	RET	OFF	RET	OFF
	CPU internal clock	ON	ON	ON	OFF	OFF	OFF	OFF	OFF	OFF	OFF
	Power state (at LPRM)	ON	ON	ON	OFF	ON	OFF	CSWRET	OFF	CSWRET	OFF
	PRCM PowerState	any		any		any		ON		RET	
	PRCM LogicRetState	any		any		any		ON		RET	
	PRCM L2MemRetState	any		any		any		ON		RET	
	PRCM ClkTrCtrl	any		any		any		HW_AUTO		HW_AUTO	
Resulting MPU/System state	Logic	ON		ON		ON		ON		ON	
	L2$	ON		ON		ON		ON		RET	
	Clock State (at PRCM)	ON		ON		ON		OFF		OFF	
	Power state (at PRCM)	ON		ON		ON		ON		CSWRET	
Measured	Time taken to go to low Power state (µs) (1)	NA		7.5		15.9		17.7		27.1	
	Time taken to wake up (us)	NA		NA (2)		3.15		5.1		6.7	

(1) Contains Software Programming overhead.

(2) Requires full boot of the system for CPU1 to come out of Forced off state.

5.1.3 Recommended Software Flow

The recommended software flow for the MPU dynamic power management is as shown in Figure 12 corresponding to Case 4.

1. The first step is to ensure the MPU C1 is put in forced-off state from the SBL.
2. The second step is a one-time initialization of the Retention parameters of the MPU subsystem and the system configuration of the MPU subsystem to allow it to go to the AUTO_CG state. This step can be done during the application initialization phase.
3. The third step to allow the MPU to go to the low-power state can be called at run time between frames in the SYSBIOS Idle Task. The MPU wake up generation is programmed each time with any newly enabled interrupts to ensure the newly enabled interrupts can wake up the MPU from retention. Note that the PMLIBCpuIdle API programs the MPU_PRCM to allow the MPUs to go to retention.

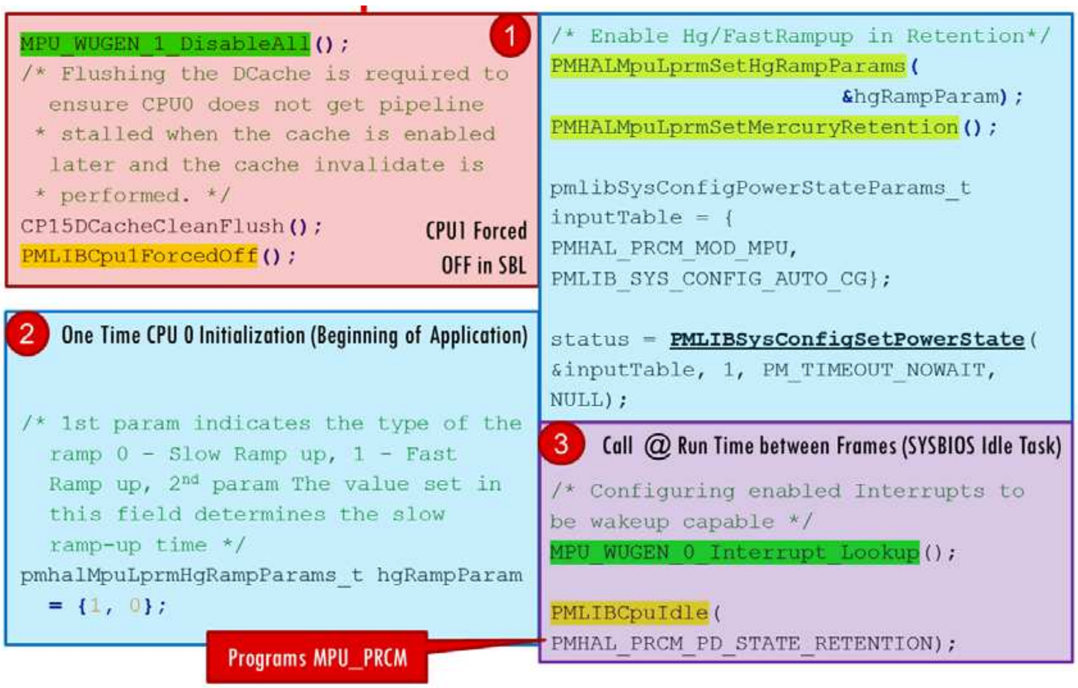


Figure 12. MPU Recommended Dynamic Power Management Software Flow

5.2 DSP Power Management

NOTE: This section is valid for TDA2x, TDA2ex and TDA3x.

5.2.1 DSP PM Blocks

The DSP performs power management with the help of the DSP Power-Down Controller (PDC) at the TMS320C66x DSP CorePac level and Sys Control/Wake up Logic at the a Subsystem level, which interacts with the SoC level Global PRCM. The Sys Control Logic is in the always on power domain.

The DSP PDC in the C66x CorePac can help enter the following power-down modes:

- During SPLOOP instruction execution L1P Memory is powered down.
- The L2 Memory is treated as Retention unTil Access (RTA) memory providing dynamic page-based automatic wake up.
- Cache control hardware is powered down when caches are disabled.
- DSP is powered down upon issuing an IDLE instruction.

- Entire that C66x Core Pac is powered down when the IDLE instruction is executed by the DSP and the PDCCMD register is programmed to put the C66x Core Pac to power down.

Further details on the DSP C66x Core Pac Power Management are shared in the *Power-Down Controller* section of the *TMS320C66x DSP CorePac User's Guide* (SPRUGW0) [5].

5.2.2 DSP PM States

This section looks at the different power states of the DSP subsystem and understands the power down and wake up latencies based on the power savings. Contact your TI representative to get access to the TDA2xx/TDA2ex Power Estimation Spread Sheet and analyze the exact power savings for the different supported low power modes.

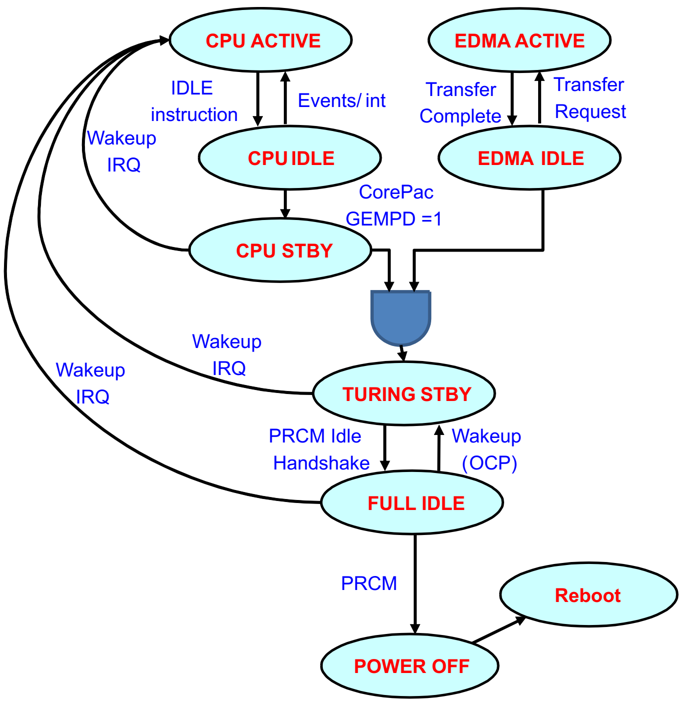


Figure 13. DSP Power States and Transitions

- ***CASE 1: DSP ON***

  In this configuration, the DSP CPU is alive and running its software. This is the highest power consumption configuration; the power consumption is determined by the kind of operations the DSP is performing. The sub-modules in the DSP subsystem are ON and the clocks are running to the Subsystem and the C66x core.

- CASE 2: DSP CPU Idle

  In this configuration, the DSP is made to go to Idle state when the DSP has nothing to run and is woken up by an event when the DSP has to resume the operation.

  1. Configure the wake up event by configuring IRQWAKEN or DMAWAKEN.
  2. DSP executes the IDLE instruction.

  The STW API that enables the DSP going into low-power mode is shown below. This API should be called from the DSP CPU.

NOTE: The input ‘pwrst’ for the below API has meaning only for MPU power-down modes. For the DSP, any valid power state can be given.

```
/**
 * \brief   Enter the CPU specified into the given low power state.
 *
 * \param   pwrst   Low power state to enter into. Refer enum
 *                  #pmhalPrcmPdState_t for details.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the desired power state was met.
 *                  PM_FAIL     If the desired power state was not met.
 */
pmErrCode_t PMLIBCpuIdle(pmhalPrcmPdState_t pwrst);
```

In this mode, while in IDLE state, if an external input interrupt source is asserted (if enabled via the DSP_SYS_IRQWAKEEN0 / DSP_SYS_IRQWAKEEN1 mask) or if an external DMA event source is asserted (if enabled via the DSP_SYS_DMAWAKEEN0 / DSP_SYS_DMAWAKEEN1 mask) or if the DSP subsystem NMI input is asserted (note that there is no wake enable mask for the non-mask able interrupt), then the DSP performs a wake up from IDLE state.

The STW APIs for initializing, enabling and disabling DSP wake up events are shown below:

```
/**
 * \brief   This API is used to initialize DSP_WUGEN. All wake up events will
 *          be disabled after initialization.
 *
 * \param   None
 *
 * \return  None.
 *
 **/
void DSP_WUGEN_IRQ_Init(void);

/**
 * \brief   This API enables the DSP wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
void DSP_WUGEN_IRQ_Enable(uint16_t intrNum);

/**
 * \brief   This API disables the DSP wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
void DSP_WUGEN_IRQ_Disable(uint16_t intrNum);
```

The Latency associated with the DSP to go to low-power state and wake up is as given in Table 13.

- ***CASE 3: DSP Subsystem Standby***

  In this configuration, the DSP is made to go to Standby state whenever the DSP has nothing to run and is again woken up by an event when the DSP has to resume processing. The DSP subsystem goes to standby state only when the EDMA transactions are not pending. This mode powers down the memory controllers in the C66x Core Pac (L1, L2, XMC, and so forth) in addition to the DSP. It is, however, a more power saving mode than DSP Idle.

  Use the following steps to put the DSP in Standby:

  1. Configure the wake up event by configuring IRQWAKEN or DMAWAKEN.
  2. Configure the DSP C66x PDCCMD register to enter Idle.
  3. Configure DSP_SYS_SYSCONFIG [5:4] STANDBYMODE to be in SMART_STANDBY_WKUP.
  4. DSP executes the IDLE instruction.

The STW API used to program the DSP C66x PDCCMD CPU software is as shown below. This API should be called from the DSP CPU. The SYSCONFIG programming to STANDBY_WKUP happens in the *PMLIBCpuIdle* function.

```
/**
 * \brief   Configure the DSP Core Pac Off mode when the DSP CPU executes
 *          Idle. Applicable only to DSP.
 *
 * \param   enable  1 - to configure the DSP PWRCMD register to put the DSP
 *                      Core Pac to off state when the DSP executes Idle
 *                      Instruction.
 *                  0 - to configure the DSP PWRCMD register to not put the
 *                      DSP CorePac to off state when the DSP executes Idle
 *                      Instruction.
 *
 * \return  none
 */
void PMLIBSetCorepacPowerDown(uint32_t enable);
```

The Latency associated with the DSP to go to low-power state and wake up is given in Table 13.

CAUTION

In order for the DSP to go into Standby, the CD_EMU clock domain should be set to SW_WKUP mode via the CM_EMU_CLKSTCTRL[1:0]CLKTRCTRL field. For more details, see the Errata ID: i879 in the Silicon Errata for TDA2x (SPRZ397) [9], TDA2ex (SPRZ428) [10] and TDA3x (SPRZ425) [11].

CAUTION

DSP internal IRQs (mapped to evt_in[31:16]) are unable to wake the DSP from a sleep/IDLE state, whereas, DSP External IRQs (from the SoC IRQ_Crossbar) (mapped to evt_in[95:32]) are able to wake the DSP. The EDMA Completion Interrupts (DSPi_IRQ_TPCC_REGION[7:0] and DSPi_IRQ_TPCC_GLOBAL) are mapped to DSP Internal IRQs; these are also provided as outputs from the DSP subsystem and are mapped as inputs to the IRQ_CROSSBAR. In order to allow the C66x DSP CorePac to wake from a low-power state when a subsystem EDMA interrupt is asserted, the desired interrupt can be mapped via the IRQ_CROSSBAR to one of the DSP External IRQs. For more details, see the Errata ID: i883 in the Silicon Errata for TDA2x (SPRZ397) [9], TDA2ex (SPRZ428) [10] and TDA3x (SPRZ425) [11].

CAUTION

The DSP Application must create a section called ".pmIdleFunc" in the DSP L2 RAM of size at least 0x80 bytes in order for the CPU IDLE function to be placed in the DSP L2 RAM. This is because XMC pre-fetches should be completed before the DSP CorePac can be powered down. If this section is not created and the DSP is CorePac is powered down, any ongoing XMC pre-fetches do not complete and the DSP might reach a possible core hang state. For more details, see the Errata ID: i898 in the Silicon Errata for TDA2x (SPRZ397) [9], TDA2ex (SPRZ428) [10] and TDA3x (SPRZ425) [11].

- ***CASE 4: DSP Subsystem Auto Clock Gated***

  In this configuration, the DSP is made to go to Standby state and the DSP Subsystem Clocks are cut whenever the DSP has nothing to run and is again woken up by an event when the DSP has to resume processing. The DSP subsystem goes to auto clock gate state only when the EDMA transactions are not pending.

Use the following steps to put the DSP in Auto Clock Gated Mode:

1. Configure the wake up event by configuring IRQWAKEN or DMAWAKEN.
2. Configure the DSP Clock Domain to be in HW_AUTO.
3. Configure the DSP C66x PDCCMD register to enter Idle.
4. Configure DSP_SYS_SYSCONFIG [5:4] STANDBYMODE to be in SMART_STANDBY_WKUP.
5. DSP executes the IDLE instruction.

An example software sequence for auto clock gating the DSP subsystem is as given in Section 5.2.3.

- ***CASE 5: DSP Subsystem Off***

  In this configuration, the DSP is made to go to power off state when the DSP is not to be used for a long time and the DSP is rebooted when the DSP is required again. This is the lowest power configuration for the DSP subsystem.

  Use the following steps to put the DSP in Power off Mode:

  1. Configure the DSP Power Domain to be in PD_OFF state.
  2. Configure the DSP Clock Domain to be in HW_AUTO.
  3. Configure the DSP C66x PDCCMD register to enter Idle.
  4. DSP executes the IDLE instruction.
  5. Application Master can then put the DSP module in disabled state once the power domain has transitioned to the power-off state.

  The sequence of STW APIs to be able to put the DSP1 to power domain off state executing from the DSP1 is shown below:

```
/* Request Power Down of DSP Power Domain */
PMHALPdmSetPDState(PMHAL_PRCM_PD_DSP1,
                   PMHAL_PRCM_PD_STATE_OFF,
                   PM_TIMEOUT_NOWAIT);

/* Request Clock Gate of the DSP Clock Domain */
PMHALCMSetCdClockMode(PMHAL_PRCM_CD_DSP1,
                      PMHAL_PRCM_CD_CLKTRNMODES_HW_AUTO,
                      PM_TIMEOUT_NOWAIT);

/* Request C66x CorePac Power Down */
PMLIBSetCorepacPowerDown(1U);

/* Configure Smart Standby Wake up and Execute WFE */
/* Input to the API is ignored for DSP */
status = PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_ON_INACTIVE);
```

  Once the DSP is in Power domain off state, the application master can execute the following module disable function to disable the DSP Subsystem.

```
PMHALModuleModeSet(PMHAL_PRCM_MOD_DSP1,
                   PMHAL_PRCM_MODULE_MODE_DISABLED,
                   PM_TIMEOUT_INFINITE);
```

CAUTION

The DSP recovery from the power-down grid OFF mode requires full boot.

For an example implementation, see the STW example in the following folder: `<STW Install Dir>\examples\pm\cpuidle`.

Table 13. DSP Power Management States and Latency

		CASE 1	CASE 2	CASE 3	CASE 4	CASE 5
Power state		**Always Enabled**	**Always Enabled**	**Always Enabled**	**Auto**	**Disabled**
Processing	Reference profile	typ/max	Idle	idle	idle	idle
	Utilization (% Active)	100	0	0	0	0
CorePac (PDCCMD) Idle Config		NA	0	1	1	1
DSP Programmed Values	PRCM PowerState	any	any	any	ON	OFF
	PRCM ClkTrCtrl	any	SW_WKUP	SW_WKUP	HW_AUTO	HW_AUTO
DSP Core State	L1 State	ON	ON	ON	ON	OFF
	L2 State	ON	RET Till Access	RET Till Access	RET Till Access	OFF
	CPU internal Clock	ON	OFF	OFF	OFF	OFF
DSP Subsystem Core State	Clock State (at PRCM)	ON	ON	ON	OFF	OFF
	Power state (at PRCM)	ON	ON	ON	ON	OFF
Measured	Time taken to go to low-power state (µs)	NA	6.5	17.2	20.75	21.8
	Time taken to wake up (µs)	NA	31.35	31.85	32.5	NA [1]

(1) The DSP recovery from the Powerdown-grid OFF mode requires full boot.

5.2.3 Recommended Software Flow

Figure 14 shows the recommended flow for the DSP to go to Auto clock gate. Note that DSP requires an additional step to put the DSP Corepac in power down, which is highlighted in yellow in Step 1.

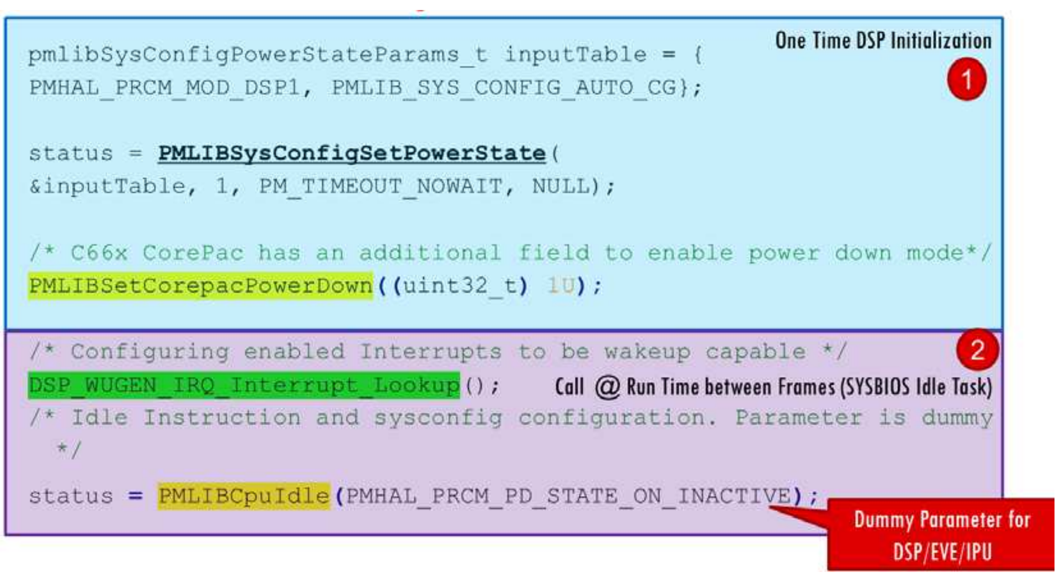


Figure 14. DSP Recommended Software Flow for DSP to go to AUTO_CG

5.3 IPU Power Management

NOTE: This section is valid for TDA2x, TDA2ex and TDA3x.

5.3.1 IPU PM Blocks

The IPUx subsystem is divided into two power domains (PD_IPU and PD_COREAON), which are controlled by the PRCM module. The PD_IPU power domain is the main power domain and includes all the IPUx subsystem components (two ARM® Cortex-M4 processors, IPUx_UNICACHE, IPUx_ROM and IPUx_RAM memories, and emulation and debug modules) except the IPUx_WUGEN.

The PD_COREAON power domain is an always on power domain. The PD_COREAON power domain contains the IPUx_WUGEN, which generates a wake-up request from external interrupts.

The IPU Voltage and Clock Domain Mapping is shown in Table 14.

Table 14. IPU1 Voltage, Power and Clock Domain Mapping

Voltage Domain	Power Domain	Functional Clock Domain	Module
VD_CORE	PD_IPU	CD_IPU	MCASP1, TIMER5, TIMER6, TIMER7, TIMER8, I2C5 (1), UART6 (1)
		CD_IPU1	IPU1

(1) Not present in TDA3x.

Further details on the IPU Power management blocks are provided in the TDA2x [2], TDA2ex and TDA3x [3] TRM.

5.3.2 IPU PM States

This section looks at the different power states of the IPU subsystem and understands the power down and wake up latencies based on the power savings. Contact your TI representative to get access to the TDA2xx/TDA2ex Power Estimation Spread Sheet and analyze the exact power savings for the different supported low power modes.

Figure 15 shows the different IPU Power states and the events which cause transition from one state to another.

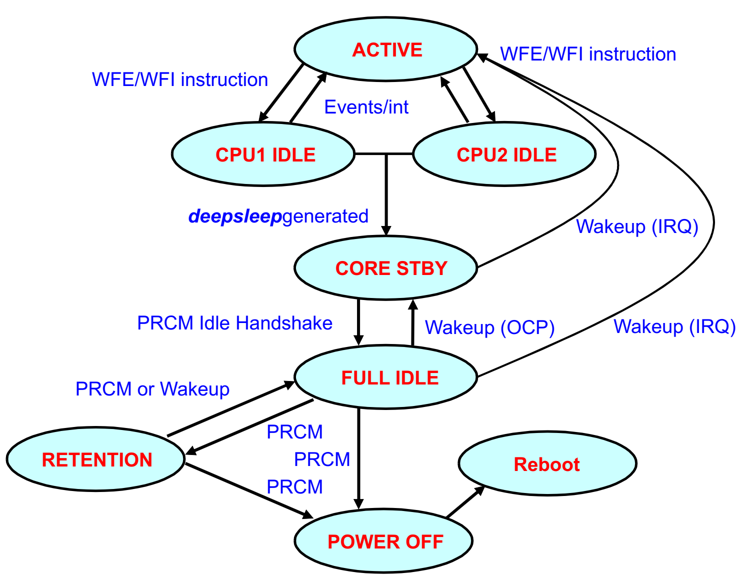


Figure 15. IPU Power States and Transitions

- ***CASE 1: IPU ON (Core 0 (C0) and Core 1 (C1) On Active)***

  In this configuration, the IPU CPU cores are alive and running their respective software. This is the highest power consumption configuration and the power consumption is determined by the kind of operations the IPU is performing. The sub-modules in the IPU subsystem are ON and the clocks are running to the subsystem and the M4 core.

- ***CASE 2: IPU CPU Idle (C0 Idle and C1 Active or C0 Active and C1 Idle)***

  In this configuration, the IPU C0 or C1 is made to go to idle state when the cores have nothing to run and are woken up by an event when the IPU has to resume the operation.

  Use the following steps to put the IPU C0 or C1 in Idle:

  1. Execute the ISB, DSB and DMB instructions to clear the M4 data and instruction pipeline.
  2. IPU C0 or C1 execute the WFI/WFE instruction.

  The STW API enables the IPU going into low-power mode by executing the sequence shown below. This API should be called from the IPU C0 to make core 0 go into Idle and from IPU C1 to make the core 1 go into Idle. Note that the power state given as an input in this API is not used and does not determine the IPU power state. In order to program the CPU to any lower-power state, program the registers using PM HAL APIs as show in the following power cases.

```
/**
 * \brief   Enter the CPU specified into the given low power state.
 *
 * NOTE:    To keep the ADAS power management simple, we only
 *          support one low-power mode per CPU. The power state is
 *          used in the API to enable supporting multiple low power
 *          states in future w/o breaking compatibility.
 *
 * \param   pwrst   Low power state to enter into. Refer enum
 *                  #pmhalPrcmPdState_t for details.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the desired power state was met.
 *                  PM_FAIL     If the desired power state was not met.
 */
pmErrCode_t PMLIBCpuIdle(pmhalPrcmPdState_t pwrst);
```

NOTE: The PMLIBCpuIdle function for Core 0 and Core 1 sets the deep-sleep mode. Therefore, when both cores are in Idle, the IPU transitions to Core Standby as discussed in Case 3 without any additional Software steps.

- ***CASE 3: IPU Subsystem Standby***

  In this power mode, both C0 and C1 are in Idle and Deep-Sleep mode is set in the System Control Register (SCR) Cortex-M4 register in the respective cores C0 and C1. The IPU asserts the standby signal to the PRCM.

  Use the following steps to put the IPU C0 and C1 in Idle:

  1. Configure the wake up event by configuring IPU Wake up Generator (IPU_WUGEN)
  2. Set the Deep Sleep bit in the System Control Register (SCR).
  3. Configure the IPU L2 MMU SYSCONFIG register to have Smart Idle as the Idle Mode.
  4. Set the IPU subsystem standby and idle modes to Smart Standby with Wake up and Smart Idle with wake up, respectively.
  5. Execute the ISB, DSB and DMB instructions to clear the M4 data and instruction pipeline.
  6. IPU C0 and C1 execute the WFI instruction.

  When both cores execute the PMLIBCpuIdle function, the IPU subsystem goes into standby state.

When the IPU is in standby, the cores can be woken up by events external to the IPU subsystem by programming the IPU_WUGEN. External events include IPU Interrupts IPU_IRQ_23 to IPU_IRQ_79. The IPU Subsystem has two registers (WUGEN_MEVT0 and WUGEN_MEVT1) that can be programmed to enable the wake up logic. Note that the WUGEN is shared between the two cores and a woken up event would wake up both cores; the core for which the event is intended would execute the ISR, while the other core remains in WFI/WFE.

In order to program the IPU_WUGEN the application software can use the following STW APIs:

```
/**
 * \brief   This API is used to initialize MPU_WUGEN. All wake up events will
 *          be disabled after initialization.
 *
 * \param   None
 *
 * \return  None.
 *
 **/
void IPU_WUGEN_Init(void);

/**
 * \brief   This API enables the IPU wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
void IPU_WUGEN_Enable(uint16_t intrNum);

/**
 * \brief   This API disables the IPU wake up event for requested interrupt.
 *
 * \param   intrNum  - Interrupt number
 *
 * \return  None.
 *
 **/
void IPU_WUGEN_Disable(uint16_t intrNum);
```

- ***CASE 4: IPU Subsystem Auto Clock Gate***

  In this configuration, the IPU is made to go to Auto Clock Gate (AUTO_CG) state and the IPU subsystem clocks are cut whenever the IPU (C0 and C1) has nothing to run and is again woken up by an event when the IPU has to resume processing.

  Use the following steps to put the IPU in AUTO_CG:

  1. Configure the IPU1 clock domain to HW_AUTO mode.
  2. Configure the wake up event by configuring the IPU wake up Generator (IPU_WUGEN)
  3. Set the Deep Sleep bit in the System Control Register (SCR).
  4. Configure the IPU L2 MMU SYSCONFIG register to have Smart Idle as the Idle Mode.
  5. Set the IPU subsystem standby and idle modes to Smart Standby with wake up and Smart Idle with wake up, respectively.
  6. Execute the ISB, DSB and DMB instructions to clear the M4 data and instruction pipeline.
  7. IPU C0 and C1 execute the WFI instruction.

  An example software sequence executed from the IPU1 C0 core is shown below. Note that the C1 is held in reset instead of having the C1 execute WFI. In case there is a valid code being run on IPU C1 once the C1 executes the PMLIBCpuIdle function, the IPU clock domain can be put into a clock-gated state.

```
/* Set IPU to Auto clock Gate */
pmlibSysConfigPowerStateParams_t inputTable = {
PMHAL_PRCM_MOD_IPU1, PMLIB_SYS_CONFIG_AUTO_CG};

status = PMLIBSysConfigSetPowerState(
&inputTable, 1U, PM_TIMEOUT_NOWAIT, NULL);

/* Assert the reset for IPU C1 */
PMHALResetAssert(PMHAL_PRCM_RG_IPU1_CPU1_RST);

/* Execute steps 3 to 6 and IPU C0 executes WFI */
/* Input to this API for IPU is dummy. Any valid value can be given */
PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);
```

- ***CASE 5: IPU Subsystem Off***

  In this configuration, the IPU is made to go to Power Domain whenever the IPU (C0 and C1) has nothing to run and the other clock domains in the power domain are clock gated.

  NOTE: In TDA3x and TDA2x/TDA2ex, IPU1 belongs to the IPU Power domain. In order to make the Power domain go to the off state, make sure the other clock domains (in the respective power domains) are also in the clock-gated state before making a successful transition to power domain off. A full IPU boot is required to become active again. All memory and logic context is lost.

  Use the following steps to put the IPU in Power Domain Off State:

  1. Configure the IPU Power domain to go to OFF state.
  2. Configure the IPU clock domain to HW_AUTO mode.
  3. Configure the other clock domains in the power domain such that they are clock gated.
  4. Set the Deep Sleep bit in the System Control Register (SCR).
  5. Configure the IPU L2 MMU SYSCONFIG register to have Smart Idle as the Idle Mode.
  6. Set the IPU subsystem standby and idle modes to Smart Standby with Wake up and Smart Idle with wake up, respectively.
  7. Execute the ISB, DSB and DMB instructions to clear the M4 data and instruction pipeline.
  8. IPU C0 and C1 execute the WFI instruction.

  An example software sequence executed from the IPU1 C0 core is shown below. Note that the C1 is held in reset, instead of having the C1 execute WFI. In case there is a valid code being run on IPU C1 once the C1 executes the PMLIBCpuIdle function, the IPU1 power domain transitions to Power domain off state.

```
/* Request IPU PD to go to Retention */
PMHALPdmSetPDState(PMHAL_PRCM_PD_IPU, PMHAL_PRCM_PD_STATE_OFF,
                   PM_TIMEOUT_NOWAIT);

/* Make sure other clock domains are in Clock Gated State */
PMHALCMSetCdClockMode(PMHAL_PRCM_CD_IPU, PMHAL_PRCM_CD_CLKTRNMODES_HW_AUTO,
                      PM_TIMEOUT_NOWAIT);

/* Request the IPU CPU clock domain is in Auto clock gated state */
PMHALCMSetCdClockMode(PMHAL_PRCM_CD_IPU1, PMHAL_PRCM_CD_CLKTRNMODES_HW_AUTO,
                      PM_TIMEOUT_NOWAIT);

/* Set the Last power domain state to on active. This is useful to check
 * once out of retention that the power domain went to retention or not */
PMHALPdmSetLastPDStatus(PMHAL_PRCM_PD_IPU, PMHAL_PRCM_PD_STATE_ON_ACTIVE);

/* Assert the reset of IPU C1 */
PMHALResetAssert(PMHAL_PRCM_RG_IPU1_CPU1_RST);

/* Execute steps 5 to 8 and IPU C0 executes WFI */
/* Input to this API for IPU is dummy. Any valid value can be given */
PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);
```

5.3.3 IPU PM TDA3x Considerations

As IPU is the boot master for TDA3x, the SoC has an additional control in the Control Module (CTRL_CORE_IPU_WAKEUP: IPU_WKUP_EN) to override the PRCM signals. By default, the IPU_WKUP_EN bit field is set high to keep the IPU1 in power domain on and override the PRCM control. The PMLIBCpuIdle API takes care of clearing the IPU_WKUP_EN bit for TDA3x.

From the Application Software one must make sure to execute the following lines of code before the IPU PMLIBCpuIdle API is executed. This makes sure that the PRCM reset states are modified to the desired values once the override is lifted.

The following code snippet assumes that the desired power-domain state and clock-domain state have already been configured based on which power state the IPU is desired to be in.

```
/*
 * This is required as the force override bit CTRL_CORE_SEC_IPU_WAKEUP
 * does not set the right values for the PRCM registers and when the
 * override is lifted then cores are left in a bad power and reset state.
 */
PMHALResetRelease(PMHAL_PRCM_RG_IPU1_CPU0_RST, PM_TIMEOUT_NOWAIT);
PMHALResetRelease(PMHAL_PRCM_RG_IPU1_RST, PM_TIMEOUT_NOWAIT);
retVal += (int32_t) PMHALModuleModeSet(PMHAL_PRCM_MOD_IPU1,
                                       PMHAL_PRCM_MODULE_MODE_AUTO,
                                       PM_TIMEOUT_NOWAIT);
```

Table 15 summarizes the different IPU power states, sub-module state in each power state and the latency to reach a given power state and the latency to exit it.

For an example implementation, see the STW example in the following folder: `<STW Install Dir>\examples\pm\cpuidle`.

Table 15. IPU Power Management States and Latency

		CASE 1	CASE 2	CASE 3	CASE 4	CASE 5
Power State		**Always Enabled**	**Always Enabled**	**Always Enabled**	**Auto**	**Disabled**
Processing	Reference profile	typ/max	Idle	idle	idle	idle
	Utilization (% Active)	100	0	0	0	0
Deep Sleep Mode		NA	0	1	1	1
IPU Programmed Values	PRCM PowerState	ON	ON	ON	ON	OFF
	PRCM ClkTrCtrl	any	SW_WKUP	SW_WKUP	HW_AUTO	HW_AUTO
IPU Core State	Core 0 Internal Clock	ON	ON	OFF	OFF	OFF
	Core 1 Internal Clock	ON	OFF	OFF	OFF	OFF
	L1 Cache	ON	ON	ON	ON	Contents Lost
	L2 Memory	ON	ON	ON	ON	Contents Lost
	WUGEN	ON	ON	ON	ON	Not Operating
IPU Subsystem Core State	Clock State (at PRCM)	ON	ON	ON	OFF	OFF
	Power state (at PRCM)	ON	ON	ON	ON	OFF
Measured	Time taken to go to low-power state (µs)	NA	2.7	2.9	3.2	3.6
	Time taken to wake up (µs)	NA	31.85	32.35	32.7	NA [(1)]

(1) The IPU recovery from the Power domain OFF mode requires full boot.

5.3.4 Recommended Software Flow

Figure 16 shows the recommended software steps to put the IPU in auto clock-gated state (CASE 4).

1. The first step is a one-time initialization to be done during the Application initialization phase to program the IPU to go to auto clock-gated state.
2. The second step is specific to the TDA3xx devices where the reset and module configurations are initialized to ensure when the PRCM override is lifted that the IPU is in the right PRCM configuration. This is also a one-time configuration.
3. The third step is an optional one-time configuration where, based on whether IPU C1 is used or not the IPU C1, reset can be asserted. If the IPU C1 is being used then code for IPU C1 should have Step 4 to make sure IPU C1 also executes the WFI instruction to allow the whole IPU subsystem to go to low-power state.
4. The fourth step looks up any newly enabled interrupts and makes them wake up capable. After the WUGEN is configured, IPU is made to go to the low-power state by calling PMLIBCpuIdle. This step can be executed between frame processing or in the SYSBIOS Idle task.

One Time IPU Initialization (1)

```
/* Set IPU to Auto clock Gate*/
pmlibSysConfigPowerStateParams_t inputTable = {
PMHAL_PRCM_MOD_IPU1, PMLIB_SYS_CONFIG_AUTO_CG};

status = PMLIBSysConfigSetPowerState(
&inputTable, 1, PM_TIMEOUT_NOWAIT, NULL);
```

Special Care about for TDA3xx (2)

```
#ifdef TDA3XX_FAMILY_BUILD
/*This is required as the force override bit CTRL_CORE_SEC_IPU_WAKEUP
 * does not set the right values for the PRCM registers and when the
 * override is lifted then cores are left in a bad power and reset state.
 */
PMHALResetRelease(PMHAL_PRCM_RG_IPU1_CPU0_RST, PM_TIMEOUT_NOWAIT);
PMHALResetRelease(PMHAL_PRCM_RG_IPU1_RST, PM_TIMEOUT_NOWAIT);
retVal += (int32_t) PMHALModuleModeSet(PMHAL_PRCM_MOD_IPU1,
                                        PMHAL_PRCM_MODULE_MODE_AUTO,
                                        PM_TIMEOUT_NOWAIT);
#endif
```

(3) Only when CPU1 not Used

```
PMHALResetAssert(PMHAL_PRCM_RG_IPU1_CPU1_RST);
```

(4) Call @ Run Time (SYSBIOS Idle Task)

```
IPU_WUGEN_Interrupt_Lookup();
retVal = (int32_t) PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);
```

Dummy Parameter for DSP/EVE/IPU

Figure 16. IPU Recommended Software Flow to Put IPU in AUTO_CG

5.4 EVE Power Management

NOTE: This section is valid for TDA2x and TDA3x.

5.4.1 EVE PM Blocks

The EVE subsystem implements two flavors of power minimization and clock gating:

- Active-Mode Power Minimization: Software transparent mode of operation, where individual leaf level IP blocks internally gate clocks dependent on the internal state of that IP being 'idle'. For example, VCOP would internally gate clocks when there is no valid program to be executed.

- Extended Duration Sleep: Explicitly requested power-down mode, handshaking with the device level PRCM, with the end result that EVE's input clocks are gated.

This application report focuses on the Extended Duration Sleep modes, which involve PRCM interaction. The EVE Voltage and Clock Domain Mapping are shown in Table 16.

Table 16. EVE Voltage, Power and Clock Domain Mapping

Voltage Domain	Power Domain	Functional Clock Domain	Module
VD_DSPEVE	PD_EVE1	CD_EVE1	EVE1
	PD_EVE2 (1)	CD_EVE2 (1)	EVE2 (1)
	PD_EVE3 (1)	CD_EVE3 (1)	EVE3 (1)
	PD_EVE4 (1)	CD_EVE4 (1)	EVE4 (1)

(1) Not present in TDA3x.

For further details on the EVE Power management blocks, see the TDA2x [2] and the TDA3x [3] TRM.

5.4.2 EVE PM States

This section looks at the different power states of the EVE subsystem and understands the power down and wake up latencies based on the power savings. Contact your TI representative to get access to the TDA2xx/TDA2ex Power Estimation Spread Sheet and analyze the exact power savings for the different supported low power modes.

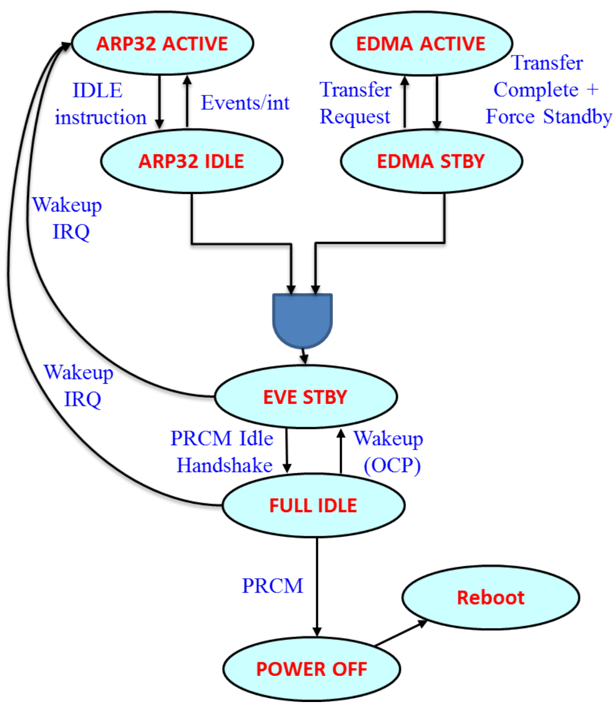


Figure 17. EVE Power States and Transitions

- ***CASE 1: EVE ON***

  In this configuration, EVE is alive and running its software. This is the highest power consumption configuration and the power consumption is determined by the kind of operations that EVE is performing. The sub-modules in the EVE subsystem are ON and the clocks are enabled for the subsystem and the ARP32 core.

- ***CASE 2: EVE CPU (ARP32) Idle***

  In this configuration, EVE ARP32 is made to go to idle state when the core has nothing to run and is woken up by an event when EVE has to resume the operation.

Use the following step to put the ARP32 in Idle:

– EVE ARP32 executes IDLE Instruction.

The STW API enables the ARP32 going into low-power mode by executing the sequence given is shown below. This API should be called from the EVE ARP32 to make EVE go into ARP32 Idle state. Note that the power state given as an input is not used and does not determine the EVE power state. In order to program EVE to any lower-power state, program the registers using PM HAL APIs as show in Case 3 and Case 4.

```
/**
 * \brief   Enter the CPU specified into the given low power state.
 *
 * NOTE:    To keep the ADAS power management simple, we only
 *          support one low-power mode per CPU. The power state is
 *          used in the API to enable supporting multiple low power
 *          states in future w/o breaking compatibility.
 *
 * \param   pwrst   Low power state to enter into. Refer enum
 *                  #pmhalPrcmPdState_t for details.
 *
 * \return  status  Returns the status of the API. This can be the following
 *                  values:
 *                  PM_SUCCESS  If the desired power state was met.
 *                  PM_FAIL     If the desired power state was not met.
 */
pmErrCode_t PMLIBCpuIdle(pmhalPrcmPdState_t pwrst);
```

- ***CASE 3: EVE Subsystem Standby***

  In this configuration, EVE is made to go to Standby state when ARP32 has nothing to run and there are no pending transactions on the EDMA. It is woken up by an event when EVE has to resume its operation.

  Use the following steps to put EVE in Standby:

  1. Configure the EVE ARP32_IRQWAKEEN to enabling waking up from standby.
  2. Ensure that all ongoing EVE EDMA transfers are complete (no pending event in EDMA CC), and configure the EVE EDMA TC0 and TC1 SYSCONFIG registers to force standby.
  3. Ensure VCOP is done with current and scheduled VLOOP processing such that VCOP asserts vec_done signal high.
  4. Configure the EVE SYSCONFIG to keep STANDBYMODE as Smart-Standby-Wkup.
  5. EVE ARP32 executes IDLE Instruction.

  The Application software can use the following STW APIs to program the ARP32_IRQWAKEEN register.

```
/**
 * \brief   This API is used to initialize ARP32_WUGEN. All wake up events will
 *          be disabled after initialization.
 * \param   None
 * \return  None.
 **/
void ARP32_WUGEN_IRQ_Init(void);

/**
 * \brief   This API enables the ARP32 wake up event for requested interrupt.
 * \param   intrNum  - Interrupt number
 * \return  None.
 **/
void ARP32_WUGEN_IRQ_Enable(uint16_t intrNum);

/**
 * \brief   This API disables the ARP32 wake up event for requested interrupt.
 * \param   intrNum  - Interrupt number
 * \return  None.
 **/
void ARP32_WUGEN_IRQ_Disable(uint16_t intrNum);
```

NOTE: EVE can wake up with the interrupts that correspond to EVE_INT00 to EVE_INT07 (see the *EVE ARP32 Interrupt Event Mapping Group0/INTC0* table in the TDA2x/TDA3x TRM) and EVE_EVT_INT[15:0] (see the *EVE ARP32 Interrupt Event Mapping Group1/INTC* table in the TDA2x/TDA3x TRM [2], [3]).

```
/* Look up the interrupts enabled and make them wake up capable */
ARP32_WUGEN_IRQ_Interrupt_Lookup();

/* Program Force Standby for the EDMA TCs */
HW_WR_REG32(SOC_EVE_EDMA_TC0_BASE + EDMA_TC_SYSCONFIG, 0x0U);
HW_WR_REG32(SOC_EVE_EDMA_TC1_BASE + EDMA_TC_SYSCONFIG, 0x0U);

/* Execute steps 5 to 6 */
/* Input to this API for EVE ARP32 is dummy. Any valid value can be given */
PMLIBCpuIdle(PMHAL_PRCM_PD_STATE_RETENTION);

/* Once Out of Standby program the EDMA TCs to have smart standby and
 * Smart Idle */
HW_WR_REG32(SOC_EVE_EDMA_TC0_BASE + EDMA_TC_SYSCONFIG, 0x28U);
HW_WR_REG32(SOC_EVE_EDMA_TC1_BASE + EDMA_TC_SYSCONFIG, 0x28U);
```

NOTE: When the ARP32 is in connected state in the debugger and executes the IDLE instruction, ARP32/EVE does not go into standby state. Once the debugger disconnects, EVE can go to standby. If the debugger is connected to ARP32 after EVE has gone into standby, then EVE is pulled out of standby state.

- ***CASE 4: EVE Subsystem Auto Clock Gated***

  In this configuration, EVE is made to go to the clock-gated state when EVE has nothing to run and is woken up by an event when EVE has to resume the operation.

  Use the following steps to put EVE in clock-gated state:

  1. Configure the EVE Clock Domain to go to HW_AUTO.
  2. Configure the EVE ARP32_IRQWAKEEN to enabling waking up from standby.
  3. Ensure that all ongoing EVE EDMA transfers are complete (no pending events in EDMA CC), and configure the EVE EDMA TC0 and TC1 SYSCONFIG registers to force standby.
  4. Ensure VCOP is done with the current and scheduled VLOOP processing such that VCOP asserts vec_done signal high.
  5. Configure EVE SYSCONFIG to keep the STANDBYMODE as Smart-Standby-Wkup.
  6. EVE ARP32 executes the IDLE Instruction.

  NOTE: When ARP32 is in connected state in the debugger and executes the IDLE instruction, ARP32/EVE does not go into IDLE state. Once the debugger disconnects, EVE can go to IDLE.

  The software sequence to make EVE go into auto clock-gated state is the same as CASE 2 (additional steps to configure the power and clock domains are given in Section 5.4.3).

- ***CASE 5: EVE Subsystem Off***

  In this configuration, EVE is made to go to Power Domain Off state when EVE has nothing to run.

  Use the following steps to put the EVE in Power Domain Off state:

  1. Configure EVE Power Domain to go to OFF.
  2. Configure EVE Clock Domain to go to HW_AUTO.
  3. Ensure that all ongoing EVE EDMA transfers are complete (no pending event in EDMA CC), and configure the EVE EDMA TC0 and TC1 SYSCONFIG registers to force standby.
  4. Ensure VCOP is done with the current and scheduled VLOOP processing such that VCOP asserts the vec_done signal high.

5. EVE ARP32 executes IDLE Instruction.

NOTE: When the ARP32 is in connected state in the debugger and executes the IDLE instruction, ARP32/EVE does not go into PD Off state. Once the debugger disconnects, then EVE can go to PD Off.

NOTE: Recovery from the PD Off state requires full EVE subsystem reboot.

The software sequence to make EVE go to the power domain off state is the same as that used in CASE 2 (using additional steps to configure the power and clock domains as shown below):

```
/* Request EVE PD to be ON and CD to be clock gated */
status = (pmErrCode_t) PMHALPdmSetPDState(PMHAL_PRCM_PD_EVE1,
                                          PMHAL_PRCM_PD_STATE_OFF,
                                          PM_TIMEOUT_NOWAIT);

status = (pmErrCode_t)PMHALCMSetCdClockMode(PMHAL_PRCM_CD_EVE1,
                      PMHAL_PRCM_CD_CLKTRNMODES_HW_AUTO,
                      PM_TIMEOUT_NOWAIT);
```

Table 17 summarizes the different EVE power states, sub-module state in each power state, the latency to reach a given power state, and the latency to exit it.

For an example implementation, see the STW example in the following folder: `<STW Install Dir>\examples\pm\arp32_cpuidle`.

Table 17. EVE Power Management States and Latency

		CASE 1	CASE 2	CASE 3	CASE 4	CASE 5
Power state		**Always Enabled**	**Always Enabled**	**Always Enabled**	**Auto**	**Disabled**
Processing	Reference profile	mac/fft/max	Idle	idle	idle	idle
	Utilization (% Active)	100	0	0	0	0
EVE Programmed Values	PRCM PowerState	ON	ON	ON	ON	OFF
	PRCM ClkTrCtrl	any	SW_WKUP	SW_WKUP	HW_AUTO	HW_AUTO
EVe Core State	ARP32 Internal Clock	ON	OFF	OFF	OFF	OFF
	EVE Subsystem Clock	ON	ON	ON	OFF	OFF
EVE Subsystem Core State	Clock State (at PRCM)	ON	ON	ON	OFF	OFF
	Power state (at PRCM)	ON	ON	ON	ON	OFF
Measured	Time taken to go to low-power state (µs)	NA	0.9	4.4	4.61	9.73
	Time taken to wake up (µs)	NA	3.8	4.15	4.4	NA [(1)]

(1) The EVE recovery from the Power domain OFF mode requires full boot.

5.4.3 Recommended Software Flow

Figure 18 shows the recommended software sequence to put EVE into auto clock-gated state (CASE 3).

1. Step 1 is a one-time initialization to configure the EVE to auto clock-gate state. This step can be done during the application initialization.
2. Step 2 can be called between frame processing or during the SYSBIOS idle Task.

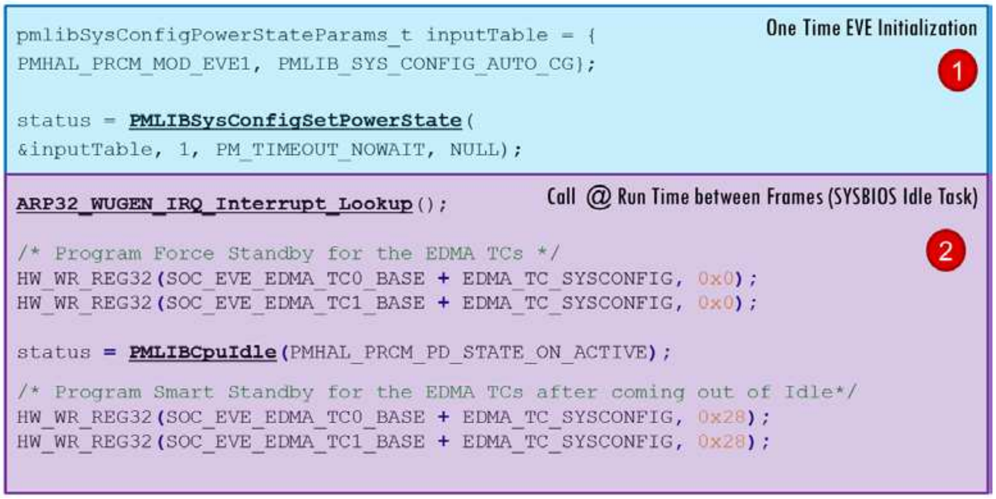


Figure 18. Recommended Software Flow for EVE to go to AUTO_CG

6 Software Thermal Management

When the device temperature heats up, due to increase in the ambient temperature and the thermal power dissipation of the device, it is important for the software to take appropriate action to lower the power consumption and to try to cool the device before the hardware mechanism of thermal shutdown kicks in. This section describes how the software can detect thermal events and help in cooling down the device when a thermal event occurs.

6.1 Temperature Management Hardware Mechanism

The TDA2xx, TDA2ex and TDA3xx devices have built-in thermal sensors that detect the junction temperature of the device between -40°C and 125°C and convert this to a 10-bit digital value through an ADC. This 10-bit value is then compared against HOT and COLD thresholds to generate HOT IRQ events when the temperature is higher that the HOT Threshold or to generate COLD IRQ events when the temperature is lower than the COLD Threshold. The device additionally has EFused temperature thresholds above which the device warm reset (TSHUT reset) is asserted. This reset is de-asserted only when the device temperature is lowered. The summarized block diagram of the on-die temperature sensor is given in Figure 19.

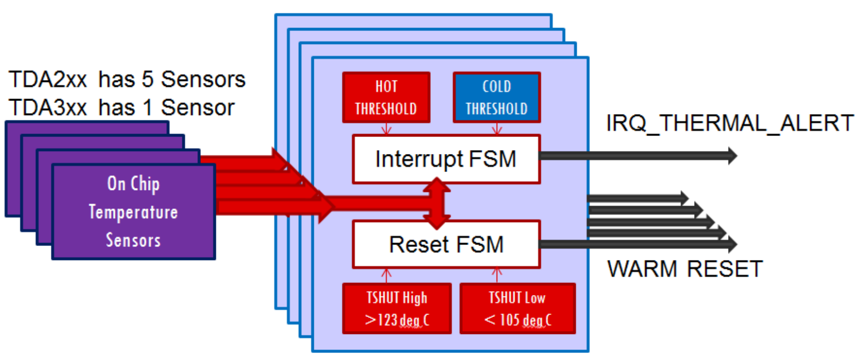


Figure 19. Functional Block Diagram of the Thermal Sensors

For more information regarding the Temperature Sensors and its registers, see the TDA2x/2ex/3x ADAS Applications Processor Technical Reference Manual ([2], [3]).

6.2 Temperature Management Software Mechanism

The basic flow of the steps that can be taken for thermal management can be visualized in Figure 20. The first step is to initialize the system to take up thermal HOT event at 100°C or any other desired high temperature during the application initialization. This makes sure that the hardware interrupts the host processor in the system when the temperature becomes higher than 100°C. When this thermal event occurs at Step 3 the appropriate cooling actions can be implemented which can bring the temperature of the device to safe operation range (70°C in the example shown in Figure 20). These cooling actions could involve changing the CPU frequency, lowering the Capture FPS by dropping frames or switching off cores altogether to have minimal application functionality at higher temperature. Once the device cools the processor would get an interrupt indicating a cold event. (Step 5) The processor can then take appropriate steps to re-enable the application to run in its full capacity.

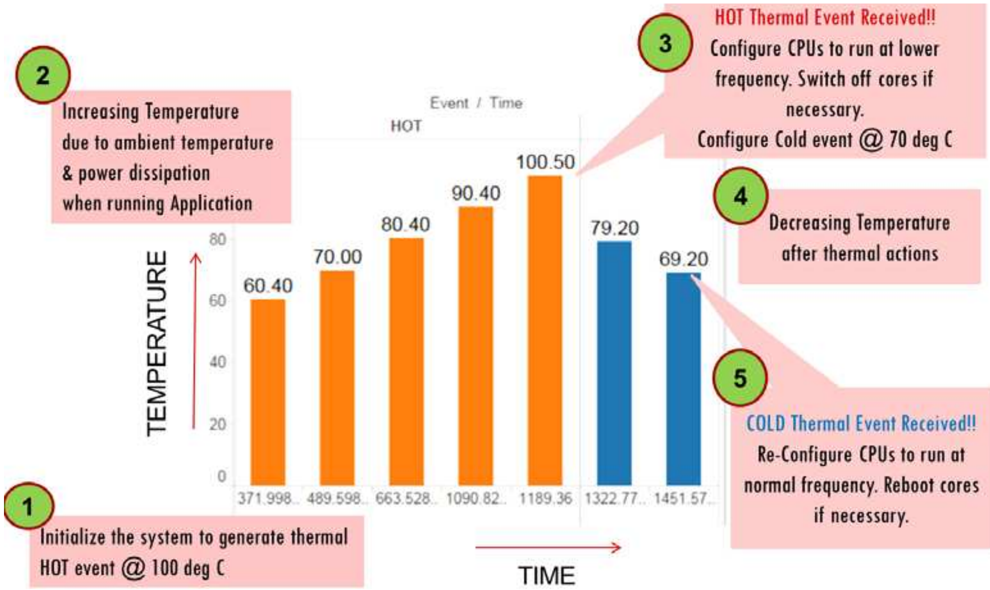


Figure 20. Thermal Management Steps

STW PM provides the following APIs to read the current temperature, set the temperature thresholds. Note that temperature reported or taken in by these APIs is in the units of milli degrees (100°C = 100 x 1000 milli °C).

```
/** \brief  Get the Band gap Temperature Sensor value for a given voltage domain
 *
 *  \param  voltId          Unique voltage domain ID. Refer enum
 *                          #pmhalPrcmVdId_t for details.
 *  \param  currTempRange   Pointer to the current temperature range. Gives the
 *                          max and min temperature which corresponds to the
 *                          read ADC value.
 *
 *  \return status          Status of the API call.
 *                          PM_SUCCESS - If the temperature is read correctly.
 *                          PM_BADARGS - If the voltage ID is invalid.
 */
int32_t PMHALBgapGetCurrTemperature(pmhalPrcmVdId_t   voltId,
                                    pmhalBgapRange_t *currTempRange);

/** \brief  This API configures the high temperature threshold for generating
 *          thermal alerts through programming the
 *          CTRL_CORE_BANDGAP_THRESHOLD_x[25:16] THOLD_HOT_x bit fields.
 *
 *  \param  voltId              Unique voltage domain ID. Refer enum
 *                              #pmhalPrcmVdId_t for details.
 *  \param  tempInMilliDegree   Temperature Threshold in milli Degree Celsius.
 *
 *  \return status              Status of the API call.
```

```
 *                              PM_SUCCESS - If the threshold is programmed
 *                                           correctly.
 *                              PM_BADARGS - If the voltage domain is not
 *                                           correct or the temperature is
 *                                           out of the range supported.
 */
int32_t PMHALBgapSetHotThreshold(pmhalPrcmVdId_t voltId,
                                 int32_t         tempInMilliDegree);

/** \brief  This API configures the Low temperature threshold for generating
 *          thermal alerts through programming the
 *          CTRL_CORE_BANDGAP_THRESHOLD_x[25:16] THOLD_COLD_x bit fields.
 *
 *  \param  voltId              Unique voltage domain ID. Refer enum
 *                              #pmhalPrcmVdId_t for details.
 *  \param  tempInMilliDegree   Temperature Threshold in milli Degree Celsius.
 *
 *  \return status              Status of the API call.
 *                              PM_SUCCESS - If the threshold is programmed
 *                                           correctly.
 *                              PM_BADARGS - If the voltage domain is not
 *                                           correct or the temperature is
 *                                           out of the range supported.
 */
int32_t PMHALBgapSetColdThreshold(pmhalPrcmVdId_t voltId,
                                  int32_t         tempInMilliDegree);
```

Figure 21 describes the software sequence that can be used to enable hot and cold events and perform thermal management in the system.

Some important points to note are the following:

- The cold event should be enabled only during the cooling down phase.
- The temperature hot event is altered in the temperature ISR to the current temperature + step size (in this case 10°C) to ensure that the processor does not keep getting interrupts while the thermal actions take effect. This also helps track the temperature events (in case the ambient temperature increases further) to make the device take multiple hot events back-to-back indicating a steadily increasing temperature.

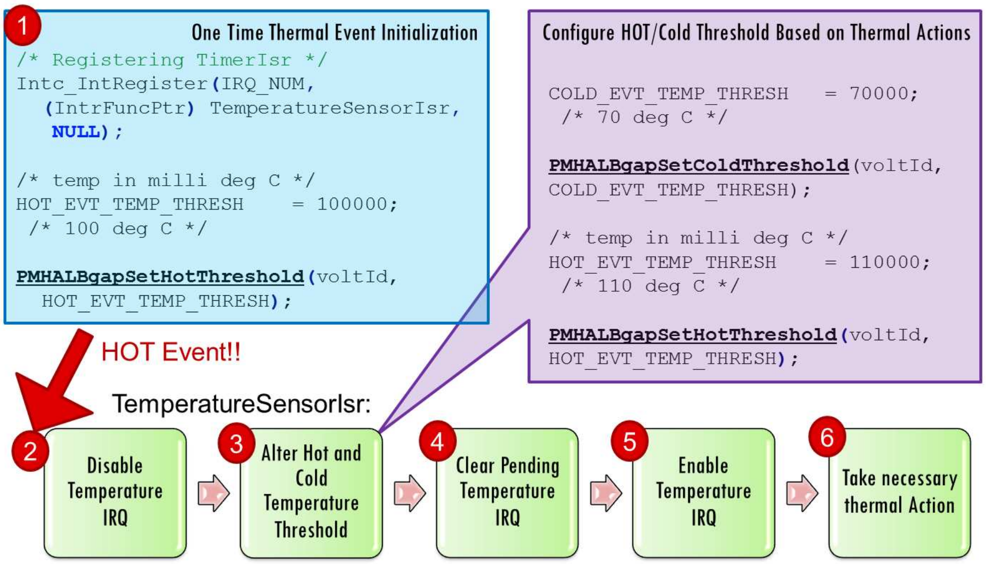


Figure 21. Thermal Management Software Sequence

- A thermal tracking system can also be created with this mechanism for modifying the temperature thresholds in the temperature ISR. Figure 22 provides an example of this thermal tracking where the device was heated and then cooled in a thermal chamber, from 40°C to 110°C then back down to 0°C.

The hot and cold thresholds were modified by 5°C on every thermal event. As the ambient temperature changed, the host processor was interrupted at the appropriate temperatures. The current temperature was recorded at every thermal event, the plot is shown in Figure 22. Note the junction temperature is higher than the ambient temperature by approximately 10°C. This difference increases as the ambient temperature increases due to higher leakage power dissipation and thermal device heat generation at higher temperatures.

- Depending on the use case operation, different regions of the device heat up slightly differently than others. Note in Figure 22 that the 500 s Time the VD_CORE is at approximately 85°C and VD_DSPEVE is at approximately 80°C. The difference in temperature is almost always <± 5-10°C. You can choose to base the thermal management on only one temperature sensor in TDA2xx/2ex to minimize software overheads.

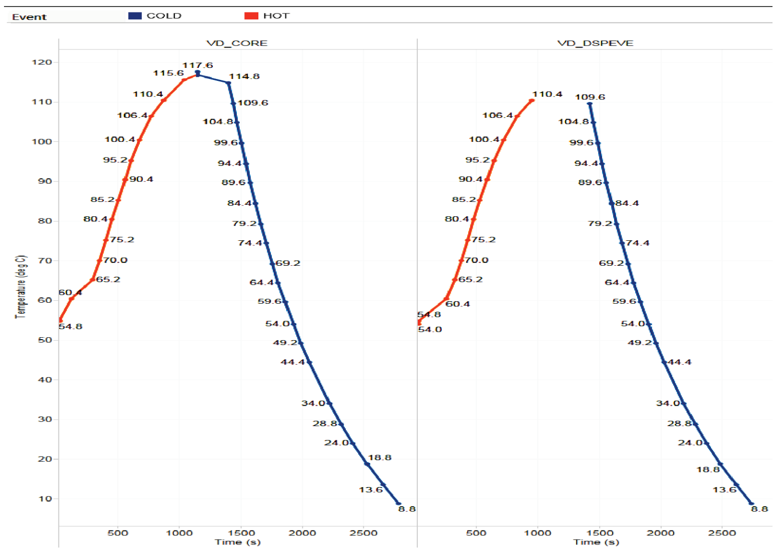


Figure 22. Thermal Tracking by Changing Thermal Thresholds in Temperature ISR

7 Reference

1. *TDA2x/TDA3xx Power Estimation Spreadsheet* (Contact your TI representative to access the spreadsheet.)
2. *TDA2x ADAS Applications Processor Technical Reference Manual* (SPRUHK5)
3. *TDA3x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.0 Technical Reference Manual* (SPRUHQ7)
4. *DRA72x SoC for Automotive Infotainment Silicon Revision 1.0 Technical Reference Manual* (SPRUHP2)
5. *TMS320C66x DSP CorePac User's Guide* (SPRUGW0)
6. *TDA2x ADAS Applications Processor 23mm Package (ABC Package) Data Manual* (SPRS859)
7. *TDA3x SoC for Advanced Driver Assistance Systems (ADAS) 15mm Package (ABF) Data Manual* (SPRS916)
8. *TDA2Ex SoC for Advanced Driver Assistance Systems (ADAS) Data Manual* (SPRS926)
9. *TDA2x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.x Silicon Errata* (SPRZ397)
10. *TDA2Ex SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.0 Silicon Errata* (SPRZ428)
11. *TDA3x SoC for Advanced Driver Assistance Systems (ADAS) Silicon Revision 1.0, 1.0A Silicon Errata* (SPRZ425)