TDA3xx Secondary Bootloader (SBL)

Rishabh Garg, Sivaraj R *ADAS Software, Processor BU*

ABSTRACT

Secondary Bootloader (SBL) is needed in order to initialize the execution environment for running the application. This includes setting up the clocks, powering on the I/O peripherals, initializing the secondary memory i.e. DDR, loading application images and booting slave cores. Advanced Driver Assist Systems (ADAS) have very specific boot requirements regarding boot time and functional safety.

This document explains the software architecture of TDA3xx Secondary Bootloader and how to use the different SBL software components in order to customize bootloader for a specific use case and software APIs to achieve the same. This is intended for designers and programmers who wish to use the SBL in their production systems.

Table of Contents

List of Figures

List of Tables

1 Secondary Bootloader (SBL) – An Introduction

The Secondary Bootloader (SBL) initializes the execution environment for multi-core application. It sets-up the AD PLL clocks to values specified in the datasheet, powers on the I/O peripherals, initializes the secondary memory i.e. DDR, loads the application images & brings-up the slave cores.

TDA3xx SBL is a single stage bootloader. ROM Bootloader (RBL) looks at the SYSBOOT switch settings to determine the boot mode. Depending on the boot mode, it initializes the peripherals and clocks which are needed by SBL. RBL jumps to the SBL start in boot media after completing its execution in case of XIP (Execute in Place) boot and in case of non-XIP boot it first copies SBL from boot media to the OCMC RAM and then jumps to the start address. TDA3xx device contains a one dual core IPU subsystem, two DSP subsystems and one EVE subsystem. Bootloader runs on the master core i.e. IPU CPU0. TDA3xx bootloader also supports Symmetric Multi-Processing (SMP) booting on dual core IPU subsystem.

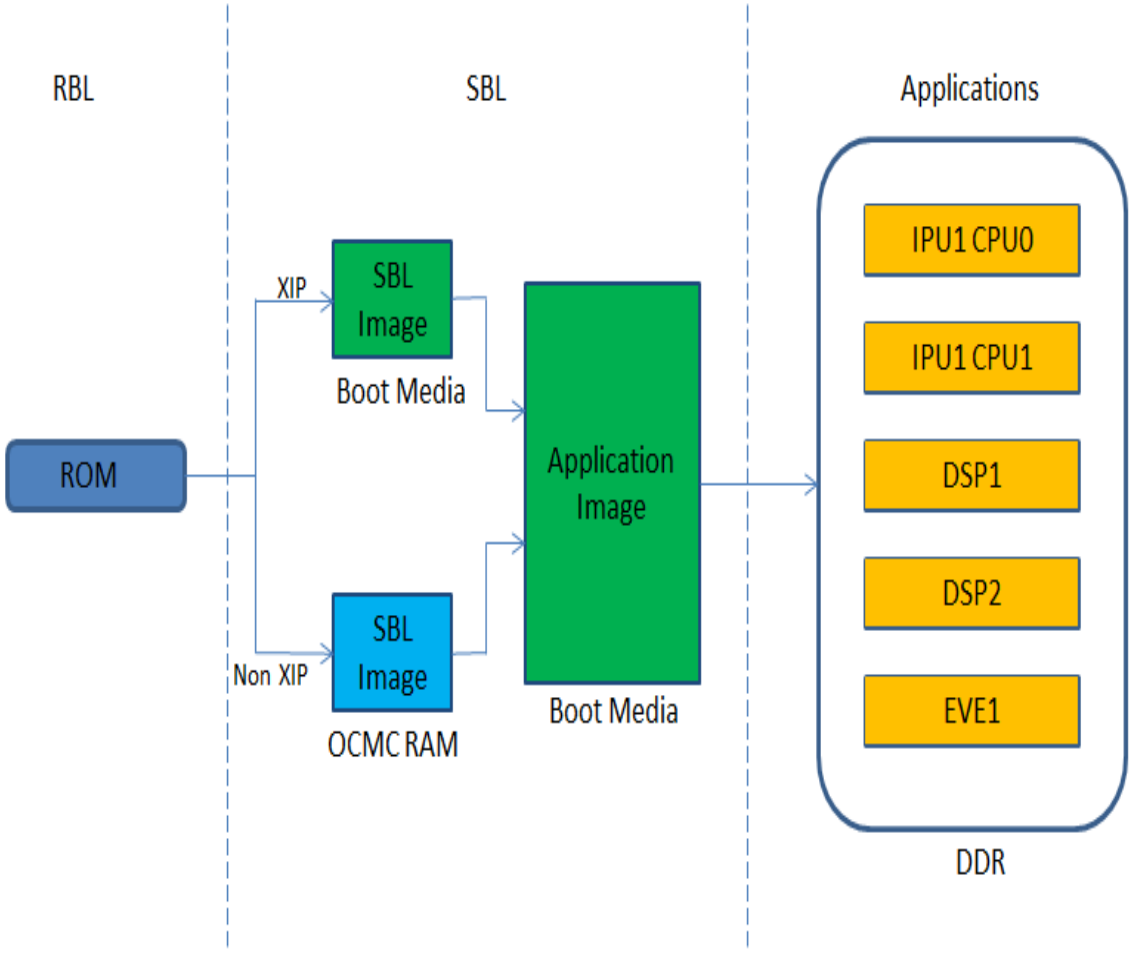

Figure 1: TDA3xx SBL Top Level Boot Flow

1.1. Supported Boot modes

TDA3xx SBL supports different boot modes depending on the SoC capabilities and requirements:

1. NOR Boot Mode: This is a XIP (Execute in Place) boot and SBL bin image and application image are both present on NOR flash.
2. SD Boot Mode: This is a non-XIP boot and SBL tiimage and application image are both present on SD card.
3. QSPI_SD Boot Mode: This is a non-XIP boot and SBL tiimage is present on QSPI flash and application image is present on SD card.

1.2. TDA3xx Bootloader Requirements

TDA3xx device is used mainly in Rear View Camera Systems (hereafter called as RVCS). There are some simple RVCS which just display the rear view on the display or intelligent ones that run Object Detection (OD) algorithms on the captured video and flash warnings for drivers. In both cases, the boot time of the system is very crucial for safety as well as overall user experience, it is always annoying to wait for system to boot up. In autonomous cars the OD result is considered to make decisions, so that is even more safety critical. Hence TDA3xx bootloader has several important safety and timing requirements. These requirements are given below:

1. Safety Requirements:
    a. Do a self-test (Logic/Memory) of various sub-systems of the SoC for integrity
    b. Verify integrity of application image by performing CRC
    c. Enable ECC on different memories in the SoC (internal and external)
2. Timing Requirements:
    a. Provide first CAN response to the main ECU in < 100 ms (including minimum safety tests)
    b. Boot the main application (including full SoC safety tests) with Video on Display in < 500 ms
    c. Boot the rest of the application (including analytics application on DSP/EVE) in < 1s

Subsequent sections explain the software architecture of TDA3xx SBL in order to meet these boot requirements.

2 SBL Software Design

Typically an ADAS RVCS use case needs few cores for the execution: 2-3 cores for running algorithms, 1-2 cores for running Safety IPs and one master core for control and SBL loads & brings up all the cores. This leads to high boot time which is undesirable. To minimize the boot time, SBL loads only the minimal needed cores (usually only master application) and other cores are loaded later on as and when needed by the master application. In order to enable this bootloader is split into three into three parts:

1. *SBL Library Layer*: This layer contains APIs needed for parsing and loading multicore App Image and booting the slave cores. The image parse logic is kept separate from data transfer logic so that any application can plug-in their own APIs. This library is used by both bootloader application and master application.

2. *SBL Utility Layer*: This layer is used only by the bootloader application and contains APIs for communicating between boot media and master core. It also contains APIs for data transfer and various wrapper APIs for interacting with other peripherals like UART, PRCM, I2C, etc.

3. *SBL Application*: This contains the bootloader application that loads and boots the application image.

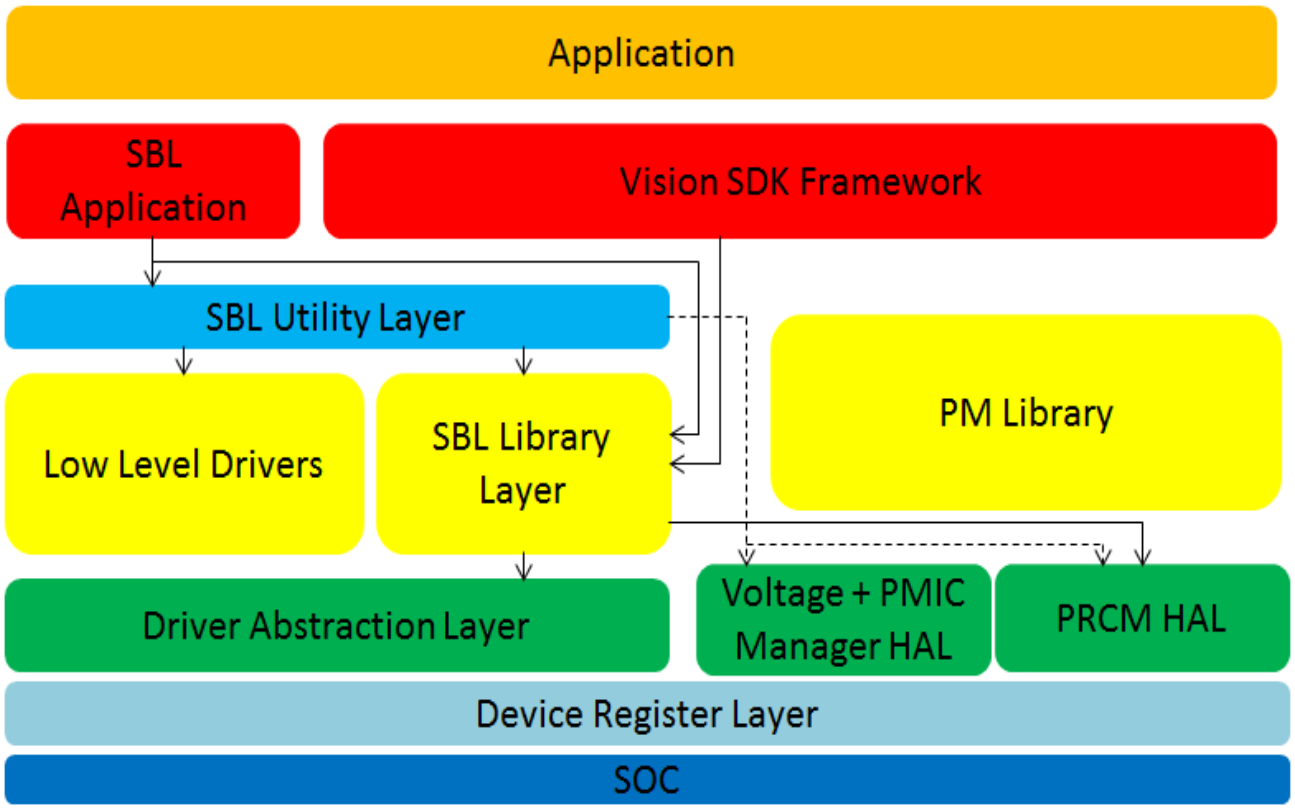

Figure 2 TDA3xx SBL Software Design

2.1 TDA3xx Bootloader Flow

With the above software design the SBL cold reset flow is described below:

1. RBL completes its execution and jumps to SBL start.

2. SBL configures DPLL CORE and DPLL PER and initializes UART. Then SBL configures the TESOC and issues IPU LBIST.

3. SBL starts again after IPU LBIST completion and checks IPU LBIST result. In case of failure boot is aborted. Otherwise SBL configures TESOC again and issues IPU MBIST.

4. SBL starts again after IPU MBIST completion. After verifying success of IPU MBIST it initializes the SOC, loads application image and boots-up the slave cores.

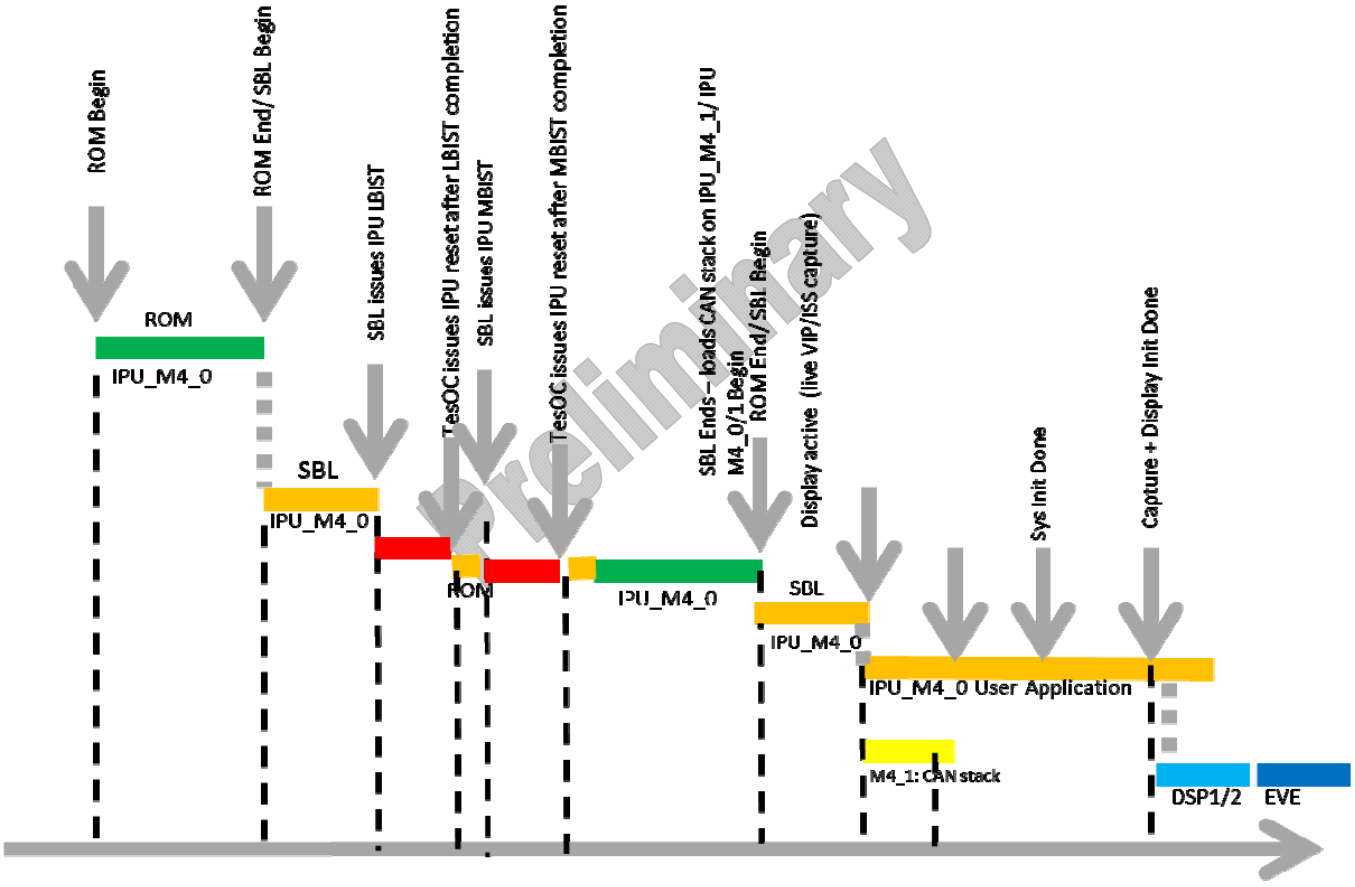

Figure 3: TDA3xx SBL Boot Flow

3 SBL - Customization

Usually ADAS use cases have very specific boot requirements due to which bootloader needs at least some level of customization for a particular use case. In the current design the same SBL image supports both silicon packages i.e. 15X15 and 12X12. Depending on the package type detected SBL takes decisions while initializing SOC. User might also want to remove this flexibility and hard code the SBL for a particular package type.

This section describes how to customize the SBL software and APIs to do the same.

3.1 UART Configuration

TDA3xx device has three UART instances: UART1, UART2 and UART3. SBL can be configured to use any UART instance for SBL prints using the below API (API takes care of PRCM and pin mux) defined in `include\sbl_utils\sbl_utils_common.h`:

```
int32_t SBLUtilsPrintfInit(UART_INST_t uartInstance);
```

UART instance enum `UART_INST_t` is defined in `include/tda3xx/soc_defines.h`. SBL Library specifies the parameter `SBL_LIB_CONFIG_UART_INST` for configuring the UART instance to be used by SBL in `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. User can specify any valid values of UART instance as per the custom use case. The underlying APIs automatically take care of configuring the Pin Mux corresponding to correct UART instance.

3.2 AVS-0 Configuration

Adaptive Voltage Scaling (AVS) Class 0 (also referred to as SmartReflex™) is a procedure for lowering the voltage on certain device power rails. AVS Class 0 attempts to normalize the power consumption across all devices. The optimal voltage for each AVS supported rail of each device is determined after analysis in the factory. This value is written in the device eFuse where it can be read through dedicated registers. These registers reside in the control module.

TDA3xx SBL configures AVS Class-0 by calling the below SBL Utility Library API defined in `include\sbl_utils\sbl_utils_common.h`:

```
int32_t SBLUtilsConfigAllVoltageRails(uint32_t oppId)
```

`oppId` refers to the operating point for which SBL needs to be built and can be set by the user at compile time. This API uses the PMIC (Power Management IC) HAL for getting the correct PMIC parameters and used the Voltage Manager (VMHAL) for setting the correct voltages. PMIC HAL can be interfaced with different PMICs as is done in TDA3xx SBL for 12x12 SVB (with LP8731) and 15x15 EVM (with TPS65917). Few APIs are defined separately for each PMIC in `pm\pmhal\pmhal_<pmic_name>.c`. In order to interface a custom PMIC with PMIC HAL, user can add a new file in which these APIs are for custom PMIC and register the same PMIC by modifying the API `SBLUtilsConfigAllVoltageRails`.

3.3 Power Domain Configuration

TDA3xx has multiple power domains which can be turned on or off depending on the use case requirements. Bootloader sets the power domain states independently and does not depend on RBL settings.

Power domain is configured using the PMHAL Power Domain Manager (hereafter called as PDM) API `PMHALPdmSetPDState()`. The signature of this API is given below:

```
int32_t PMHALPdmSetPDState(pmhalPrcmPdId_t pdId,
                           pmhalPrcmPdState_t pdState,
                           uint32_t timeout);
```

Enumerations `pmhalPrcmPdId_t` and `pmhalPrcmPdState_t` are defined in the header file `include/tda3xx/pmhal_prcm.h`.

Structure `sblutilsPowerDomainStateConfig_t` used for configuring power domain configuration is defined in `include/sbl_utils/sbl_utils_common.h`. The definition of this structure is given below:

```
typedef struct sblutilsPowerDomainStateConfig
{
    uint32_t    domainId;
    /**< Power Domain Id, refer to enum #pmhalPrcmPdId_t for values */
    uint32_t    powerDomainState;
    /**< State for which power domain should be configured,
     *   refer to enum #pmhalPrcmPdState_t for values
     */
    const char *powerDomainName;
    /**< Power domain name */
} sblutilsPowerDomainStateConfig_t;
```

Wrapper API `SBLUtilsConfigPowerDomains()` is used by SBL application to configure clock domains. This API loops over the table `gPowerDomainStateTable` defined in `bootloader\sbl_utils\src\sbl_utils_tda3xx.c` which is essentially an array of `sblutilsPowerDomainStateConfig_t` structures. The power domain states specified in the table correspond to the states supported by TDA3xx device and use case requirements.

User can modify this table and turn power domains on/off as per the use case requirements using correct enum values for domain id and domain state from corresponding PRCM header file `pmhal_prcm.h`.

3.4 DPLL Configuration

Each SOC has its own set of DPLLs used to provide the clock to various peripherals. There are five DPLLs available on TDA3xx device.

Table 1 DPLL List for TDA3xx Device

Sr. No.	DPLL Name
1.	DPLL_CORE
2.	DPLL_PER
3.	DPLL_DDR
4.	DPLL_GMAC_DSP
5.	DPLL_EVE_VID_DSP

Out of these five DPLLs, TDA3xx bootloader configures only the first four DPLLs (for OPP NOM and OD, OPP High configures all five DPLLs as explained in section OPP Configuration) using Power Management (hereafter called as PM) Clock Manager (hereafter called as CM) HAL API `PMHALCMDpllConfigure()` defined in the PM HAL file `include/pm/pmhal/pmhal_cm.h`. `DPLL_EVE_VID_DSP`'s configuration is done by the application in case of OPP NOM and OPP OD. The signature of the API is given below:

```
int32_t PMHALCMDpllConfigure(pmhalPrcmNodeId_t dpllId,
                             const pmhalPrcmDpllConfig_t *pDpllData,
                             uint32_t timeOut);
```

The structure `pmhalPrcmDpllConfig_t` and parameter `dpllId` are defined in `include/tda3xx/pmhal_prcm.h`. The structure contains multiplier (m); divider (n) and an array of post divider elements i.e. structure `pmhalPrcmPllPostDivValue_t`. The definition of these structures is given below:

```
typedef struct pmhalPrcmDpllConfig
{
    uint16_t multiplier;
    /**< Multiplier value for DPLL configuration. */
    uint16_t divider;
    /**< Divider value for DPLL configuration. */
    uint32_t dutyCycleCorrector;
    /**< DutyCycleCorrector enable or disable */
    pmhalPrcmPllPostDivValue_t *postDivConfList;
    /**< List of post divider configuration values. */
```

```
        uint32_t maxPostDivElems;
        /**< Maximum postdivider elements. */
    } pmhalPrcmDpllConfig_t;

    typedef struct pmhalPrcmPllPostDivValue
    {
        pmhalPrcmDpllPostDivId_t postDivId;
        /**< Unique Id of the post divider. */
        uint32_t configValue;
        /**< Division value. */
    } pmhalPrcmPllPostDivValue_t;
```

The DPLL structures for post dividers and DPLL configuration are defined in the file `bootloader/sbl_lib/src/sbl_lib_tda3xx_prcm_dpll.c`. SBL first gets the DPLL configuration structure using API `SBLLibGetDpllStructure()` defined in `include/sbl_lib/sbl_lib.h`. Then it is passed to API `PMHALCMDpllConfigure()`. The signature of `SBLLibGetDpllStructure()` API is given below:

```
    int32_t SBLLibGetDpllStructure(uint32_t dpllId,
                                   uint32_t sysClk,
                                   uint32_t opp,
                                   pmhalPrcmDpllConfig_t **dpllCfg);
```

The parameter `sysClk` passed to this API is derived by calling Clock Manager API `PMHALCMGetSysClockFreqEnum()` and `opp` is the Operating Point for which this DPLL should be configured. OPP values are defined in enum `sbllibPrcmDpllOpp_t` present in `include/sbl_lib/sbl_lib.h` file.

3.4.1 Example : DPLL Core Configuration

In order to configure DPLL CORE SBL application follows the below steps:

1. SBL first gets the SYSCLK value by calling API `PMHALCMGetSysClockFreqEnum()`.
2. SBL calls API `SBLLibGetDpllStructure()` with the required parameters. This returns the DPLL configuration structure.
3. SBL calls API `PMHALCMDpllConfigure()` in order to set this configuration.

The sequence is given below:

```
sysClkFreq = PMHALCMGetSysClockFreqEnum();
retVal = SBLLibGetDpllStructure(PMHAL_PRCM_DPLL_CORE,
                                sysClkFreq,
                                SBLLIB_PRCM_DPLL_OPP_NOM,
                                &dpllParams);
retVal += PMHALCMDpllConfigure(PMHAL_PRCM_DPLL_CORE,
                               dpllParams,
                               PM_TIMEOUT_INFINITE);
```

3.4.2 Modifying DPLL Configuration

DPLL configuration can be modified as given below:

1. Modifying Multiplier (m) and Divider (n)

In file `bootloader\sbl_lib\src\sbl_lib_tda3xx_prcm_dpll.c` DPLL configuration structure `pmhalPrcmDpllConfig_t` has an instance defined for a particular DPLL, SYSCLK frequency, OPP and silicon package type. User needs to modify this definition in case he wants to modify only the multiplier and divider elements.

E.g. DPLL Core has configuration structure named `dpllCore15X15CfgOppNom_20` for 15X15 silicon package, OPP NOM and 20 MHz SYSCLK with m value as 266 and n as 4 which should be modified to make the above modification.

2. Modifying Post Divider Elements

Similar to DPLL configuration structure there is an array of post divider element structures that is defined for a particular DPLL, SYSCLK frequency, OPP and silicon package type. This array has a variable number of structures depending on the number of post divider elements that needs to be set. User can modify the values of post dividers and add or remove structures depending on the post dividers used in the system.

E.g. DPLL Core has a post divider element array `dpllCore15X15PostDivCfgOppNom_20` for 15X15 silicon package, OPP NOM and 20 MHz SYSCLK value which should be modified.

3. Using Different Input Clock

In case the user is using a different input clock he needs to add new structures for DPLL configuration and post divider elements. Then he needs to add a switch case for the new SYSCLK value in the API `SBLLibGetDpllStructure()` for getting the correct DPLL configuration structures. Below convention is followed for naming DPLL configuration structure and post divider array and user should follow the same.

Configuration Structure: `<dpllName><PackageType>Cfg<OPP>_<SYSCLK>`

Post Divider Array: `<dpllName><PackageType>PostDivCfg<OPP>_<SysclkValue>`

3.5 Clock Configuration

TDA3xx has PRCM IP that provides multi-level clocks which can be enabled or disabled depending on the use case requirements. In order to enable clock for a particular module first the clock domain under which the module falls needs to be enabled and then the module needs to be enabled. In order to disable a module clock module can be turned off and in order to disable a clock domain all modules in that particular clock domain need to be turned off.

3.5.1 Clock Domain Configuration

Clock domain is configured using the PMHAL CM API `PMHALCMSetCdClockMode()`. The signature of this API is given below:

```
int32_t PMHALCMSetCdClockMode(pmhalPrcmCdId_t clockdomainId,
                              pmhalPrcmCdClkTrnModes_t mode,
                              uint32_t timeOut);
```

Enumerations `pmhalPrcmCdId_t` and `pmhalPrcmCdClkTrnModes_t` are defined in `include/tda3xx/pmhal_prcm.h`.

Structure `sblutilsClockDomainStateConfig_t` used for configuring clock domain configuration is defined in `include/sbl_utils/sbl_utils_common.h`. The definition of this structure is given below:


```
typedef struct sblutilsClockDomainStateConfig
{
      int32_t     domainId;
      /**< Clock Domain Id, refer to enum #pmhalPrcmCdId_t for values */
      uint32_t    clockDomainState;
      /**< State for which clock domain should be configured,
       *   refer to enum #pmhalPrcmCdClkTrnModes_t for values
       */
      const char *clockDomainName;
      /**< Clock domain name */
} sblutilsClockDomainStateConfig_t;
```

Wrapper API `SBLUtilsConfigClockDomains()` is used by SBL application to configure clock domains. This API loops over the table `gClockDomainStateTable` defined in `bootloader\sbl_utils\src\sbl_utils_tda3xx.c` which is essentially an array of `sblutilsClockDomainStateConfig_t` structures. The clock domain states specified in the table correspond to the states supported by TDA3xx device and use case requirements.

User can modify this table and enable/disable clock domains as per the use case requirements using correct enum values for clock domain id and clock domain state from corresponding pmhal_prcm.h.

3.5.2 Module Clock Configuration

Module clock is configured using the PMHAL Module Manager (hereafter called as MM) API PMHALModuleModeSet(). The signature of this API is given below:

```
int32_t PMHALModuleModeSet(pmhalPrcmModuleId_t moduleId,
                           pmhalPrcmModuleSModuleMode_t modMode,
                           uint32_t timeOut);
```

Enumerations pmhalPrcmModuleId_t and pmhalPrcmModuleSModuleMode_t are defined in include/tda3xx/pmhal_prcm.h.

Structure sblutilsModuleEnableConfig_t used for configuring module configuration is defined in include/sbl_utils/sbl_utils_common.h. The definition of this structure is given below:

```
typedef struct sblutilsModuleEnableConfig
{
    uint32_t    moduleId;
    /**< Module Id, refer to enum #pmhalPrcmModuleId_t for values */
    uint32_t    moduleEnableMode;
    /**< Mode in which module should be configured,
     *   refer to enum #pmhalPrcmModuleSModuleMode_t for values
     */
    const char *moduleName;
    /**< Module name */
} sblutilsModuleEnableConfig_t;
```

Wrapper API SBLUtilsConfigModules() is used by SBL application to configure module clocks. This API loops over the table gModuleEnableTable defined in bootloader\sbl_utils\src\sbl_utils_tda3xx.c which is essentially an array of sblutilsModuleEnableConfig_t structures. The module clock states specified in the table correspond to the states supported by TDA3xx device and use case requirements.

User can modify this table and enable/disable module clocks as per the use case requirements using correct enum values for module id and module state from corresponding pmhal_prcm.h.

3.6 DDR Configuration

TDA3xx device supports different types of DDR and different DDR speeds depending on the silicon package type and the use case requirements. List of different DDR types along with their speed supported by the bootloader is given below:

Table 2 Supported DDR Types

Sr. No.	DDR Type	DDR Speed
1.	DDR3	532 MHz 400 MHz 333 MHz
2.	DDR2	400 MHz 333 MHz
3.	LPDDR2	400 MHz 333 MHz

SBL Library specifies various DDR configuration parameters that are defined in `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. These parameters are:

1. `SBL_LIB_CONFIG_DDR_TYPE`
2. `SBL_LIB_CONFIG_DDR_SPEED`
3. `SBL_LIB_CONFIG_DDR_TYPE_12X12`
4. `SBL_LIB_CONFIG_DDR_SPEED_12X12`

First two parameters for valid for 15X15 silicon package and the other two are valid for 12X12 silicon package. The valid values that are currently supported are present in the interface file `include/sbl_lib/sbl_lib.h`. SBL application first detects the silicon package and then depending on the silicon package it picks up the correct DDR type and speed to be set for that particular package. In order to hard code SBL for a particular package user can specify required values for DDR type and speed for specific package and invalid value for the other package. In case user wants to define a custom use case he first needs to define a new SBL Library configuration in `bootloader/sbl_lib/sbl_lib_config_tda3xx.h` (User can refer to `SBL_CONFIG_CUSTOM1` as an example). Similar changes need to be made in `build/makerules/component.mk` makefile for addition of custom use case.

3.6.1 Modifying Timing Parameters

Note: Ideally User should not try to change the timing parameters. Proper analysis of various aspects like effect on silicon life, stability should be done before making the changes. SBL Utility layer defines different APIs for configuring DDR depending on the DDR type in `include/sbl_utils/sbl_utils_tda3xx_ddr_config.h`.

```
void SBLUtilsDDR3Config(uint32_t ddrSpeed);

void SBLUtilsDDR2Config(uint32_t ddrSpeed);

void SBLUtilsLPDDR2Config(uint32_t lpddr2Speed);
```

These APIs are called depending on the DDR configuration parameters as defined in the above section. DDR configuration is basically a four step process:

1. Configure DDR IOs present in Control Module which is same for a particular DDR type.
2. Set reset value of local EMIF parameter variables which is usually same for a particular DDR type for different speeds
3. Set timing value of local EMIF parameter variables depending on the DDR speed
4. Configure the EMIF registers.

Below example demonstrates the API sequence for configuring DDR3 at 532 MHz:

```
TDA3xx_CM_DDRIO_Config(CFG_MODE_DDR3);
if ((uint32_t) SBLLIB_DDR_SPEED_532MHZ == ddrSpeed)
{
      TDA3xx_reset_emif_params_ddr3_532(SOC_EMIF1_CONF_REGS_BASE);
      TDA3xx_set_emif_params_ddr3_532(SOC_EMIF1_CONF_REGS_BASE);
}
EMIF_Config(SOC_EMIF1_CONF_REGS_BASE,
            CFG_MODE_DDR3,
            HW_LEVELING_ENABLED,
            ENABLE_ECC);
```

API TDA3xx_set_emif_params_ddr3_532() needs to be modified if user wants to set a new DDR speed. Similar sequences are present for DDR2 and LPDDR2 at the same location.

3.7 AMMU Configuration

IPU subsystem contains L1 MMU (AMMU) for the L1 Cache (also called unicache) and thus, labelled also as IPU_UNICACHE_MMU. It provides the cache with region-based address translation, read/write control, access type control, and multilevel cache maintenance. The IPU_UNICACHE_MMU regions are register entry based to enable low-latency lookup and it is not a table walking MMU. IPU_UNICACHE_MMU contains total 16 entries and its configuration is given below:

Table 3 IPU_UNICACHE_MMU Configuration

Number of large pages	4
Size of large pages	512 MB or 32 MB (configurable)
Number of medium pages	2
Size of medium pages	256 KB or 128 KB (configurable)
Number of small pages	10
Size of small pages	16 KB or 4 KB (configurable)

RBL configures the AMMU as given below (empty entries are not shown in the tables):

Table 4 RBL AMMU Config for Small Pages

Small Pages (x10)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x5500_0000	0x0000_0000	16 KB
1	0x5508_0000	0x4000_0000	16 KB
2	0x5500_4000	0x0000_4000	16 KB
3	0x5502_0000	0x0002_0000	16 KB
4	0x5502_4000	0x0002_4000	16 KB
5	0x5502_8000	0x0002_8000	16 KB
6	0x5502_C000	0x0002_C000	16 KB

Table 5 RBL AMMU Config for Medium Pages

Medium Pages (x2)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x4030_0000	0x0030_0000	256 KB
1	0x4034_0000	0x0034_0000	256 KB

Table 6 RBL AMMU Config for Large Pages

Large Pages (x4)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x0800_0000	0x0800_0000	32 MB
1	0x5C00_0000	0x0400_0000	32 MB

RBL wakes up the IPU subsystem i.e. both CPU cores IPU_C0 and IPU_C1 and hands over the control to SBL on master core i.e. IPU_C0 whereas the second core is held in "WFE" state. SBL wakes up the IPU_C1 core by sending "SEV" instruction before reconfiguring AMMU.

Note: This is a must as SBL reconfigures 0x0 to IPU RAM instead of IPU ROM and hard fault gets created on IPU_C1 in case IPU_C1 is woken up after reconfiguring AMMU.

SBL calls the below function to configure AMMU which is defined in the header file `include\sbl_utils\sbl_utils_tda3xx.h`:

```
void SBLUtilsConfigIPU1DefaultAMMU(void);
```

SBL provides an interface for customizing AMMU configuration in form of various macros that are defined in `bootloader\sbl_utils\src\sbl_utils_tda3xx_ammu_config.h`. User can modify these macros to set up different AMMU configuration.

3.7.1 Present AMMU Configuration

TDA3xx SBL configures AMMU as given in below tables:

Table 7 SBL AMMU Config for Small Pages

Small Pages (x10)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x4330_0000	0x6330_0000	16 KB
1	0x5508_0000	0x4000_0000	16 KB
2	0x4330_4000	0x6330_4000	16 KB
3	0x4208_1000	0x6208_1000	16 KB
4	0x4208_B000	0x6208_B000	16 KB
5	0x4208_C000	0x6208_C000	16 KB
6	0x4883_A000	0x6883_A000	16 KB
7	0x420A_0000	0x620A_0000	16 KB
8	0x4208_6000	0x6208_6000	16 KB

Table 8 SBL AMMU Config for Medium Pages

Medium Pages (x2)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x4030_0000	0x0030_0000	256 KB
1	0x5502_0000	0x0000_0000	128 KB

Table 9 SBL AMMU Config for Large Pages

Large Pages (x4)			
Page Num	**Physical Address**	**Virtual Address**	**Size**
0	0x0800_0000	0x0800_0000	32 MB
1	0x4000_0000	0x4000_0000	512 MB
2	0x8000_0000	0x8000_0000	512 MB
3	0x8000_0000	0xA000_0000	512 MB

3.7.2 Disabling AMMU in SBL

TDA3xx SBL provides an option where user can disable the AMMU configuration set by SBL at the end of its execution so that application can set up its AMMU independently. This option is disabled in SBL by default and can be enabled by setting the macro `SBL_LIB_CONFIG_DISABLE_AMMU` to 1 in `bootloader\sbl_lib\sbl_lib_config_tda3xx.h`. However there are a few factors that should be kept in mind while doing so as explained below.

TDA3x SBL supports different optimization levels: high, medium and low. When SBL is built with optimization mode high, it gives the control to IPU_C1 application as soon as it loads the image on IPU_C1 whereas in lower optimization levels the control is given after booting up all other slave cores at the end just before giving control to IPU_C0 application. This leads to two conclusions:

1. User cannot disable AMMU in case of Optimization Level High.
2. In lower optimization levels, IPU_C1 image should have a handshake mechanism with IPU_C0 so that Core1 knows when the AMMU has been set up by Core0 in order to avoid invalid access.

Hence it is recommended that user should set up the AMMU in SBL and not disable it in the end.

3.8 Cache Configuration

IPU subsystem supports 32 KB L1 Cache called `IPU_UNICACHE` divided into 16 banks. There are no separate Instruction (I) or Data (D) caches. The cache ability is provided through the AMMU present in the cache i.e. whether a particular CPU access needs to be cached or not depends on the corresponding AMMU policy register.

TDA3xx SBL enables cache before giving control to the master application for all boot modes. On the other hand TDA3xx SBL enables cache while executing itself only for QSPI and NOR boot modes (usually used in production) and does not enable cache for QSPI_SD boot mode (used only for development). SBL calls the API `SBLUtilsEnableIPU1Unicache` defined in `include/sbl_utils/sbl_utils_tda3xx.h` in order to enable the Unicache.

TDA3xx SBL provides an option where user can choose to disable the Unicache at the end of SBL execution. By default, SBL does not disable Unicache before giving control to IPU application. This can be done by setting the macro `SBL_LIB_CONFIG_DISABLE_UNICACHE` to 1 in `bootloader\sbl_lib\sbl_lib_config_tda3xx.h`.

As explained in previous section, IPU Core1 boot up first in case of optimization level high and if the user chooses to disable unicache (IPU Core1 application is already running at the point of disabling Unicache), it may lead to random behavior on IPU Core1 application. Hence it is recommended that user should not disable Unicache at end of SBL.

3.9 Pin Mux Configuration

TDA3xx SBL configures the pin mux required for accessing UART, Boot Media and Ethernet i.e. RGMII. It uses the APIs present in the `utils_platform` library for doing the pin mux configuration. Below sections describe the changes needed in order to customize bootloader for a custom use case:

3.9.1 UART Pin Mux

As explained in section 3.1 SBL calls `SBLUtilsPrintfInit()` for initializing UART. This API further calls the `utils_platform` API `PlatformUARTSetPinMux()` defined in `platform/platform.h` for doing pin mux. The signature of this API is given below:

```
int32_t PlatformUARTSetPinMux(UART_INST_t num);
```

UART instance enum `UART_INST_t` is defined in `include/tda3xx/soc_defines.h`. Depending on the UART instance, `PlatformUARTSetPinMux()` API calls another API `PlatformUART<instance_num>SetPinMux()` defined in `platform/platform.h`. E.g. for UART1, `PlatformUART1SetPinMux()` API is called. In case any UART instance is connected differently as compared to TI EVM, user should look at the schematics of the custom board and modify the API `PlatformUART1SetPinMux()` accordingly for UART1 and similar APIs for UART2 and UART3.

3.9.2 Boot Media Pin Mux

TDA3xx SBL supports three boot modes: QSPI, QSPI_SD and NOR. Depending on the boot mode it configures the pins needed for accessing boot media in order to read application images. For configuring pin mux for TI EVM, SBL calls a wrapper API `SBLConfigTIEVMPad()` defined in `bootloader/sbl/src/sbl_tda3xx_priv.h`. This API further calls different API for doing pin mux depending on the boot mode.

1. QSPI Boot

In QSPI boot application image is flashed on QSPI flash and in order to access the flash pin mux is configured by calling the API `PlatformQSPISetPinMux()` defined in `platform/platform.h`. In case QSPI flash is connected differently as compared to TI EVM, user should look at the schematics of the custom board and modify the API `PlatformQSPISetPinMux()` accordingly.

2. QSPI_SD Boot

In QSPI_SD boot application image is present on the SD card and in order to access the SD card pin mux is configured by calling the API `PlatformMMCSDSetPinMux()` defined in `platform/platform.h`.

MMC instance enum `MMC_INST_t` is defined in `include/tda3xx/soc_defines.h` and there is only once valid instance on TDA3xx device i.e. MMC4. `PlatformMMCSDSetPinMux()`calls another API `PlatformMMCSD4SetPinMux()` defined in `platform/platform.h` for configuring pin mux for MMC4.

In case MMCSD instance is connected differently as compared to TI EVM, user should look at the schematics of the custom board and modify the API `PlatformMMCSD4SetPinMux()` accordingly.

3. NOR Boot

In NOR boot application image is flashed on NOR flash and in order to access the flash pin mux is configured by calling the API `PlatformGPMCSetPinMux()` defined in `platform/platform.h`. In case NOR flash is connected differently as compared to TI EVM, user should look at the schematics of the custom board and modify the API `PlatformGPMCSetPinMux()` accordingly.

3.9.3 RGMII Pin Mux

TDA3xx SBL configures pin mux for RGMII0 by calling the API `PlatformRGMIISetPinMux()` defined in `platform/platform.h`. On TDA3xx device MMC and RGMII require same pins to be configured. Thus RGMII pin mux configuration is done after copying App Image in QSPI_SD boot-mode. This is done by calling the API `SblConfigRGMIIPad()`. In case RGMII0 is connected differently as compared to TI EVM, user should look at the schematics of the custom board and modify the API `PlatformRGMIISetPinMux()` accordingly.

3.10 Slave Core Configuration

TDA3xx device has one dual core IPU subsystem out of which IPU_C0 is master core and two single core DSP subsystems & one single core EVE subsystem (both DSPs and EVE are slave cores). SBL parses and loads the application images for all the cores (master and slave both) and boots up the slave cores. SBL also supports SMP booting for dual core IPU subsystem.

SBL parses the multicore application image and stores the entry point for each application image in the structure instance `sblEntryPoints` which is of type `sbllibEntryPoints_t` defined in `include/sbl_lib/sbl_lib.h`. The definition of this structure is:

```
typedef struct sbllibEntryPoints
{
    uint32_t entryPoint[SBLLIB_MAX_ENTRY_POINTS];
    /**< Entry point for different cores:
     * Array Index  0: MPU CPU0
     * Array Index  1: MPU CPU1
     * Array Index  2: IPU1 CPU0
     * Array Index  3: IPU1 CPU1
     * Array Index  4: IPU2 CPU0
     * Array Index  5: IPU2 CPU1
     * Array Index  6: DSP1
     * Array Index  7: DSP2
     * Array Index  8: EVE1
     * Array Index  9: EVE2
     * Array Index 10: EVE3
     * Array Index 11: EVE4
     */
} sbllibEntryPoints_t;
```

Here `SBLLIB_MAX_ENTRY_POINTS` is the sum of all the cores present in the device. E.g. TDA3xx can have maximum five entry points i.e. IPU1 CPU0, IPU1 CPU1, DSP1, DSP2 and EVE1. Remaining entries are invalid.

Application image load API identifies if the application image has an image for a particular core and assigns the entry point accordingly (assigns same entry point to IPU1 CPU0 and IPU1 CPU1 in case of SMP booting). After identifying the entry points SBL brings up the slave cores using below APIs defined in `include/sbl_lib/sbl_lib.h`:

```
void SBLLibDSP1BringUp(uint32_t entryPoint, uint32_t sblBuildMode);
void SBLLibDSP2BringUp(uint32_t entryPoint, uint32_t sblBuildMode);
void SBLLibEVE1BringUp(uint32_t entryPoint, uint32_t sblBuildMode);
```

SBL build mode is identified as per user input at compile time. This is explained in detail in next section.

3.10.1 SBL Build Mode

SBL supports a build feature called build mode. There are two possible build modes:

1. Production Build Mode (Prod)

2. Development Build Mode (Dev)

In production build mode, if a valid app image for a given CPU core is not found, then the SBL puts the corresponding core to "PD Off" State which reduces leakage power. In development build mode, the CPU is not put to power down mode when valid app image is not there. This can be set by the user by specifying SBL_BUILD_MODE parameter at compile time: "dev" for development and "prod" for production.

User can modify the Slave core boot-up APIs to hard code SBL for a particular build mode. Although not recommended user can also modify these APIs to change the slave core boot-up sequence if there are specific reasons to do so.

EVE MMU needs to be configured before EVE1 bring up. This is is done in the below API defined in bootloader/sbl_lib/src/sbl_lib_tda3xx_platform.c:

```
static void SblLibEVEMMUConfig(uint32_t baseAddr, uint32_t entryPoint);
```

This API maps System MMU for EVE's entry point, L4 peripheral space and EVE's internal memories. It is expected that DDR mappings will be done by the application. User can add additional MMU entries in this API as per the use case requirements.

IPU Core1 boot-up is a special case as described in detail in next section.

3.10.2 IPU Core1 Boot-up

SBL defines a section ".ipu1_1_init" in its memory map and places the API SblIPU1Core1Init at the beginning of this section. This API is defined in bootloader/sbl/src/sbl_tda3xx_main.c:

```
    void SblIPU1Core1Init(void);
```

This API waits in a while loop till the time valid entry point is written by SBL at the location SBLLIB_IPU1_CORE_1_ENTRY_POINT_ADDRESS. Once SBL writes the entry point address, the API reads it and jumps to entry point address. In case multicore application image does not contain a valid image for IPU Core1, SBL writes the valid address value SBLLIB_IPU1_CORE1_BOOT_ADDRESS_NO_IMAGE at this location. These address macros are defined in the file include/sbl_lib/sbl_lib_tda3xx.h.
Before sending "SEV" instruction to wake-up IPU Core1, SBL programs AUXBOOT registers in such a way that IPU Core1 jumps to this memory section and hence the API SblIPU1Core1Init(). Thereafter, Core1 jumps to the entry point address as populated by SBL.

User might want to modify these macros depending on the system memory map. Similar changes should be made to SBL memory map as well.

3.11 App Image Load Configuration

SBL calls the API `SBLLoadAppImage()` in order to parse and load the multi-core application image from boot media to DDR memory and core specific internal memories. This API is defined in file `bootloader/sbl/src/sbl_tda3xx_priv.h`:

```
int32_t SBLLoadAppImage(sbllibAppImageParseParams_t *imageParams);
```

Structure `imageParams` contains the application parse parameters as defined by the SBL application layer. Definition of this structure is given below:

```
typedef struct sbllibAppImageParseParams
{
    sbllibMetaHeader_t      *appImgMetaHeader;
    /**< Multi-core application image meta header */
    sbllibRPRCImageHeader_t *appImgRPRCHeader;
    /**< Multi-core application RPRC header */
    sbllibEntryPoints_t     *entryPoints;
    /**< Structure containing entry points for different cores */
    uint32_t                 appImageOffset;
    /**< Multi-core image offset in Boot Media */
    uint32_t                 ddrAppImageAddress;
    /**< DDR location where RPRC image is copied before parsing */
    uint32_t                 enableCrc;
    /**< Whether to enable CRC on Multi-core application image */
}sbllibAppImageParseParams_t;
```

Structure members `appImgMetaHeader` and `appImgRPRCHeader` represent the multi-core application image Meta header and RPRC header respectively. User can refer to SBL user guide for details of these structures.

`entryPoints` structure contains the entry points for different cores as explained in section "`Slave Core Configuration`".

`appImageOffset` contains the offset address where the multi-core application image is flashed. TDA3xx SBL defines macros `SBLUTILS_APP_IMAGE_OFFSET_QSPI` and `SBLUTILS_APP_IMAGE_OFFSET_NOR` in `include/sbl_utils/sbl_utils_common.h` for defining offset in QSPI and NOR boot mode respectively. Offset is 0 for QSPI_SD mode.

TDA3xx SBL supports CRC check on application image. Due to this the application image has to be transferred twice from boot media: first time for CRC calculation and second time for loading the application to DDR run address. Here the boot media file transfer throughput becomes a bottle neck. The approach taken by SBL is to copy application image from boot media to DDR reserved area (one RPRC image at a time) and then copy from DDR to calculate CRC and load to DDR run address. This utilizes DDR bandwidth and minimizes boot time impact. `ddrAppImageAddress` defines the DDR reserved area address.
`enableCrc` defines whether the CRC check should be done on application image before loading.

User needs to make sure that multicore application image does not contain any loadable section in reserved DDR area and can assign values to `imageParams` as per his system level requirements.

API `SBLLoadAppImage()` calls various APIs depending on the boot mode for loading the application image as explained in section "`Boot Media Configuration`". These APIs further call an API `SBLLibRegisterImageCopyCallback()` for registering the image copy callback functions. The signature of this API is given below:

```
int32_t SBLLibRegisterImageCopyCallback(SBLLibMediaReadFxn mediaReadFxn,
                                        SBLLibDDRReadFxn   ddrReadFxn,
                                        SBLLibSeekFxn      seekFxn);
```

These function prototypes are defined in `include/sbl_lib/sbl_lib.h` and are given below:

```
typedef int32_t (*SBLLibMediaReadFxn)(void    *dstAddr,
                                      uint32_t srcAddr,
                                      uint32_t length);
typedef int32_t (*SBLLibDDRReadFxn)(void       *dstAddr,
                                    const void *srcAddr,
                                    uint32_t    length);
typedef void (*SBLLibSeekFxn)(uint32_t *srcAddr,
                              uint32_t location);
```

`SBLLibMediaReadFxn` is the prototype of data copy function used to read from boot media while loading application image. `SBLLibDDRReadFxn` is the prototype of data copy function used to read from DDR while loading application image. `SBLLibSeekFxn` is the prototype of seek function used while loading image.

User can choose to define custom functions for loading application image and register them using the `SBLLibRegisterImageCopyCallback()` API.

3.12 Boot Media Configuration

As explained in section "App Image Load Configuration", API SBLLoadAppImage() calls different APIs for loading application image depending on boot mode. These APIs first configure the boot media for enabling read access depending on the boot mode and media. Below section explains settings done for each boot mode:

1. QSPI Boot

TDA3xx SBL calls the API SBLUtilsQspiBootRprc()for loading the application image in QSPI boot mode. This API is defined in include/sbl_utils/sbl_utils_common.h.
The signature of this API is given below:

```
int32_t SBLUtilsQspiBootRprc(sbllibAppImageParseParams_t *imageParams);
```

This API further calls an API QSPI_Initialize() in order to initialize the QSPI flash. This API is defined in qspilib/qspi_flash/qspi_flash.h and the signature is given below:

```
uint32_t QSPI_Initialize(qspi_DeviceType_e DeviceType);
```

SBL Library specifies QSPI flash type as parameter SBL_LIB_CONFIG_QSPI_FLASH_TYPE which is defined in bootloader/sbl_lib/sbl_lib_config_tda3xx.h. User can modify this parameter to select the QSPI flash as per their custom EVM in case he wants to use QSPI boot mode.

2. QSPI_SD Boot

TDA3xx SBL calls the API SBLUtilsQspiSDBootRprc()for loading the application image in QSPI_SD boot mode. This API is defined in include/sbl_utils/sbl_utils_common.h.
The signature of this API is given below:

```
int32_t SBLUtilsQspiSDBootRprc(sbllibAppImageParseParams_t *imageParams)
```

This API further calls various StarterWare board library APIs in order to initialize the board mux for accessing SD card. User might need to modify these APIs depending on their board layout.

3. NOR Boot

TDA3xx SBL calls the API SBLUtilsNorBootRprc()for loading the application image in NOR boot mode. This API is defined in include/sbl_utils/sbl_utils_common.h.
The signature of this API is given below:

```
int32_t SBLUtilsNorBootRprc(sbllibAppImageParseParams_t *imageParams)
```

As NOR is XIP boot SBL configures NOR timing parameters at the beginning rather than here. SBL execution start from the label sbl_nor_start defined in the file bootloader/sbl/src/sbl_m4_init_nor.asm.
GPMC IP is used as the interface to NOR flash. SBL defines seven timing parameters macros in the same file with macro names being SPANSION_NOR_GPMC_CONFIG_x.

User can refer to the custom NOR flash spec for the timing parameters and TDA3xx TRM for GPMC register details and modify these macros accordingly.

3.13 TESOC Configuration

TDA3xx SBL ensures functional safety of device by performing various tests using Safety IPs. TESOC (Tester on Chip) is used to do a self-test (Logic/Memory) of various sub-systems of the SoC. TDA3xx SBL performs TESOC tests on the following domains: DSP1, DSP2, EVE, ISS, VIP and DSS. SBL calls the API `SblConfigAndInitiateOtherTesocTests()` for running TESOC tests on above mentioned domains. This API is defined in the file: `bootloader/sbl/src/sbl_tda3xx_main.c.`
SBL Library specifies a configuration parameter to enable these TESOC tests: `SBL_LIB_CONFIG_ENABLE_OTHER_TESOC_TESTS`. This parameter is defined in the file `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. User can set this parameter to 0 in case he wants to disable the TESOC tests.

3.14 CRC Configuration

TDA3xx SBL supports CRC check on the application image to verify its integrity. SBL Library specifies a configuration parameter to enable the CRC check on application image: `SBL_LIB_CONFIG_ENABLE_CRC`. This configuration parameter is defined in the file `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. User can set this parameter to 0 in case he wants to disable the CRC check on application image. If this parameter is set to 1, application image load API calculates the CRC and performs the check. For performing the CRC check, SBL registers a callback function for calculating CRC by calling `SBLLibRegisterCRCCallback()` API. This API is defined in `include/sbl_lib/sbl_lib_tda3xx.h` and the signature is given below:


```
int32_t SBLLibRegisterCRCCallback(SBLLibCalculateCRCFxn calculateCRCFxn);
```

The function prototype is defined in `include/sbl_lib/sbl_lib_tda3xx.h` and is given below:

```
    typedef int32_t (*SBLLibCalculateCRCFxn)(const void     *srcAddr,
                                             uint32_t        crcDataSize,
                                             crcSignature_t *crcSignVal);
```

`SBLLibMediaReadFxn` is the prototype of SBL function used for calculating CRC. User can choose to define a custom function for calculating CRC and register it using the callback API `SBLLibRegisterCRCCallback()`. It should be taken care that is this registration is done before calling the API to load application image.

3.15 ECC Configuration

TDA3xx SBL enables ECC on both internal and external memories i.e. DDR and IPU RAM. Below section describes how to enable/disable or modify ECC for both memories:

1. External Memory

SBL Library specifies a configuration parameter to enable the ECC for DDR memory `SBL_LIB_CONFIG_ENABLE_EMIF_ECC`. This configuration parameter is defined in the file `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. User can set this parameter to 0 in case he wants to disable the ECC for DDR memory. If this parameter is set to 1, there are a few more parameters that should be specified by user for ECC to be enabled correctly. Firstly the user needs to specify the DDR start address by specifying the parameter `SBL_LIB_CONFIG_EMIF_START_ADDR` (usually 0x80000000U). For enabling ECC user can specify two address ranges and ECC can be enabled either inside or outside the address range in any combination. Also as same SBL image can support both 12X12 and 15X15 silicon package, different macros are defined for each silicon type. User also needs to define the size of the memory connected on the board. Set of parameters for 15X15 package is given below:

i. SBL_LIB_CONFIG_EMIF_SIZE_15X15
ii. SBL_LIB_CONFIG_EMIF_ECC_START_ADDR1_15X15
iii. SBL_LIB_CONFIG_EMIF_ECC_END_ADDR1_15X15
iv. SBL_LIB_CONFIG_EMIF_ECC_REG1_RANGE_TYPE_15X15
v. SBL_LIB_CONFIG_EMIF_ECC_START_ADDR2_15X15
vi. SBL_LIB_CONFIG_EMIF_ECC_END_ADDR2_15X15
vii. SBL_LIB_CONFIG_EMIF_ECC_REG2_RANGE_TYPE_15X15

SBL Utility layer defines the API to configure ECC `SBLUtilsEmifECCConfigure()` in `include/sbl_utils/sbl_utils_common.h` and the signature is given below:

```
int32_t SBLUtilsEmifECCConfigure(void);
```

This API should be modified only after careful analysis.

When ECC is enabled, user needs to make sure that correct interrupt handlers are registered by the application for both one bit and two bit errors.

2. IPU RAM

SBL Library specifies a configuration parameter to enable the ECC for IPU RAM memory `SBL_LIB_CONFIG_ENABLE_IPU_RAM_ECC`. This configuration parameter is defined in the file `bootloader/sbl_lib/sbl_lib_config_tda3xx.h`. User can set this parameter to 0 in case he wants to disable the ECC for IPU RAM. SBL pre-fills the IPU RAM by using EDMA and cache maintenance operations.

SBL defines the API to configure ECC for IPU RAM `SblConfigIpuRamECC()` in `bootloader/sbl/src/sbl_tda3xx_main.c` and the signature is given below:

```
static void SblConfigIpuRamECC(void);
```

This API should be modified only after careful analysis. When ECC is enabled, user needs to make sure that correct interrupt handlers are registered by the application for both one bit and two bit errors.

3.16 OPP Configuration

TDA3xx device supports multiple operating points (OPPs) at which it can be operated. The frequencies at which different clocks can operate are given below for different OPPs for 15X15 silicon package:

Table 10 Supported OPP and Max Frequency

Description	OPP NOM	OPP OD	OPP High
	Max Freq (MHz)	Max Freq (MHz)	Max Freq (MHz)
DSP_CLK	500	709	745
EVE_FCLK	500	667	667

It should be noted that frequencies at different OPPs are not yet defined for 12X12 package.

In TDA3xx device EVE and DSPs (both DSP1 and DSP2) can derive their clock from different DPLLs depending upon the system requirements. The clocking structure is explained in detail in below sections.

3.16.1 EVE Clock Configuration

EVE can derive its clock from any one of the three DPLLs `DPLL_EVE_VID_DSP`, `DPLL_GMAC_DSP` and `DPLL_CORE` as shown in figure below:

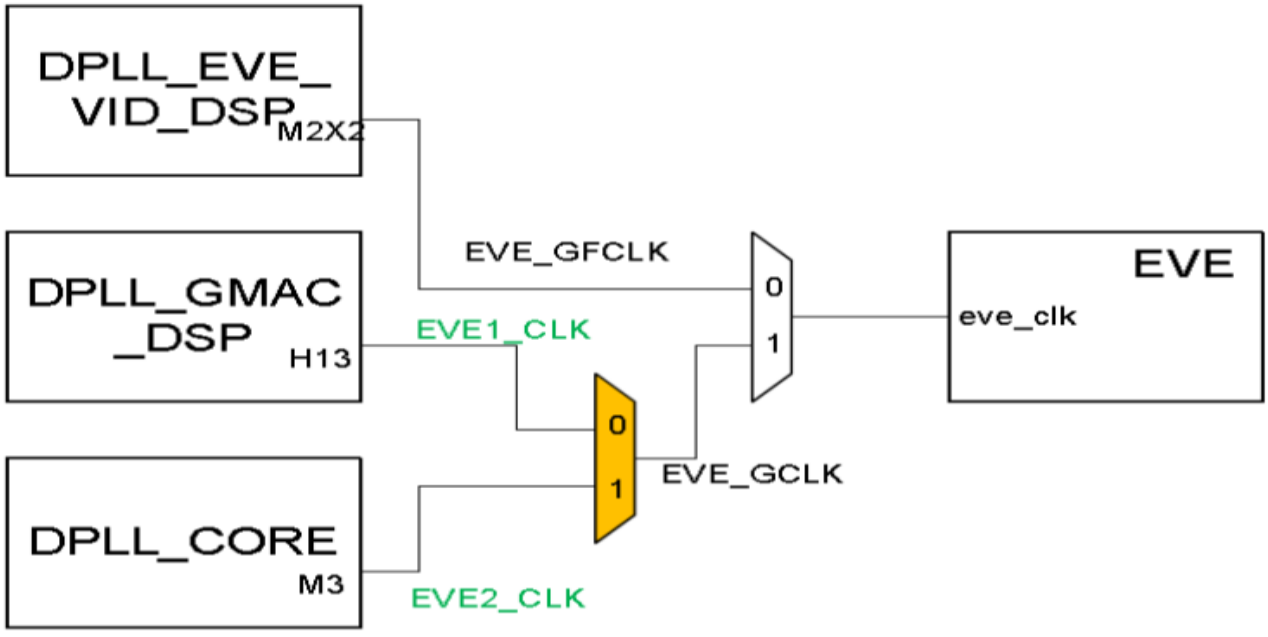

Figure 4 EVE Clocking Options

Firstly selection needs to be made between EVE1_CLK and EVE2_CLK as shown in figure. This can be done by calling the PM API `PMHALCMMuxParentSelect()`. The signature of this API is given below:

```
int32_t PMHALCMMuxParentSelect(pmhalPrcmNodeId_t muxId,
                               pmhalPrcmNodeId_t pParentNodeId);
```

For selecting the required clock between EVE1_CLK and EVE2_CLK, mux id is `PMHAL_PRCM_MUX_EVE_GCLK_MUX` and parent node id could be either `PMHAL_PRCM_DPLL_DSP_GMAC` or `PMHAL_PRCM_DPLL_CORE`.

Similarly for selecting between EVE_GFCLK and EVE_GCLK the same API needs to be called. In this case the mux id is `PMHAL_PRCM_MUX_EVE_CLK_MUX` and the parent node id could be `PMHAL_PRCM_MUX_EVE_GCLK_MUX` or `PMHAL_PRCM_DPLL_EVE_VID_DSP`.

3.16.2 DSP Clock Configuration

Both DSP instances can derive clock from any one of the three DPLLs `DPLL_EVE_VID_DSP`, `DPLL_GMAC_DSP` and `DPLL_CORE` as shown in figure below:

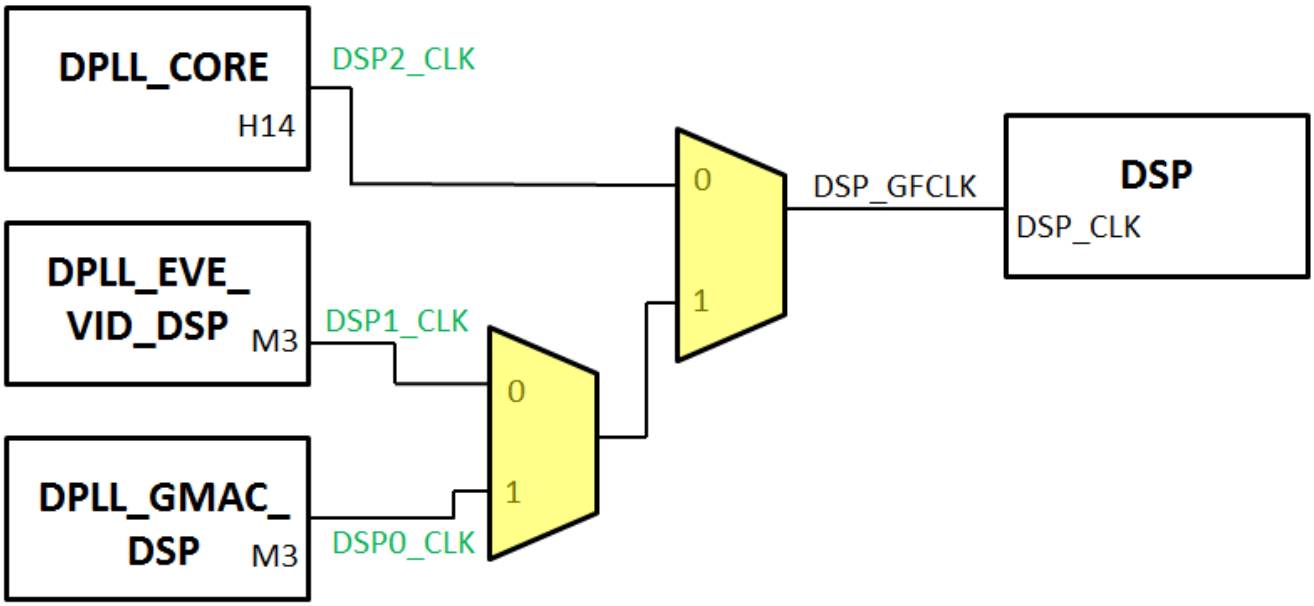

Figure 5 DSP Clocking Options

Selection can be made between DSP0_CLK, DSP1_CLK and DSP2_CLK as shown in figure. This can be done by calling the PM API `PMHALCMMuxParentSelect()`. Mux id is `PMHAL_PRCM_MUX_DSP_GFCLK_MUX` and depending on the frequency requirements parent node id could be `PMHAL_PRCM_DPLL_DSP_GMAC`, `PMHAL_PRCM_DPLL_CORE` or `PMHAL_PRCM_DPLL_EVE_VID_DSP`.

3.16.3 OPP Configuration in SBL

SBL can be built for different OPPs by specifying command line parameter "OPPMODE". For different OPPs SBL sources DSP and EVE clocks as given below:

1. For OPP NOM, both EVE and DSP clock is sourced from DPLL_DSP_GMAC.
2. For OPP OD, DSP clock is sourced from DPLL_CORE and EVE clock is sourced from DPLL_DSP_GMAC.
3. For OPP High, DSP clock is sourced from DPLL_EVE_VID_DSP and EVE clock is sourced from DPLL_DSP_GMAC.

User can modify the divider values as described in detail in section "DPLL configuration" in order to derive a different frequency value other than used in TI SBL. However various system constraints need to be kept in mind while setting the dividers as each DPLL provides clock to many modules and device does not have dedicated DPLLs for EVE and DSP. One such particular case is OPP High Configuration as described in next section.

3.16.4 OPP High Configuration

The maximum frequency for OPP high as described in device datasheet is 745 MHz but this frequency can be achieved only by sourcing clock from DPLL_EVE_VID_DSP. DPLL_CORE and DPLL_DSP_GMAC have other modules connected to them and no optimum combination is possible for setting OPP High using these two DPLLs. When DSP clock is sourced from DPLL_EVE_VID_DSP, it limits the possible values of VID_PIX clock and hence limits the frequency at which display can be run. As per the current configuration there are only two frequencies at which display can be run which are 74.5 MHz (for LCD) and 149 MHz (for HDMI). User needs to understand this constraint and hence design the system accordingly.

TDA3xx SBL defines an additional parameter SBL_LIB_CONFIG_DSP_OPP_HIGH_750MHZ in bootloader/sbl_lib/sbl_lib_config_tda3xx.h that needs to be set to '1' by user in order to enable OPP High in SBL. This parameter is set to '0' by default.

References

1. *TDA3xx Technical Reference Manual Version H. (SPRUHQ7H) Chapter 24*
2. *ADAS StarterWare User Guide*
3. *ADAS SBL User Guide*
4. *TDA3xx Datasheet Manual. (SPRS916D)*