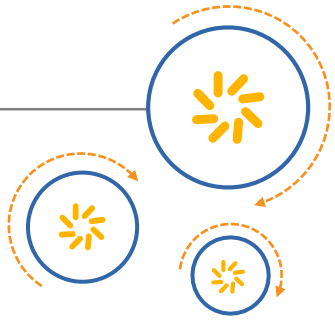
Qualcomm Technologies, Inc.



Modem Subsystem GPIO Software

Application Note

80-NL239-3 E

December 3, 2015

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	December 2013	Initial release
B	October 2014	Updated the following to include MSM8936/MSM8939 and MSM8909 information: ▪ Document title ▪ Sections 1.1, 1.2, 2.1, 4.1.2, and 5.3
C	April 2015	Updated the document title; updated Sections 1.1, 2.1, 3.1, and 4.1.1
D	November 2015	Updated section 1.1 and 2.1
E	November 2015	Updated section 1.1, 2.1,4.1.1, 4.1.4, 4.3 and 5.3

Contents

Figures

Tables

1 Introduction

1.1 Purpose

This document describes the AMSS89x6, AMSS89x9, AMSS8952, AMSS8976, AMSS8937 and AMSS8953 modem subsystem GPIO customizations made to suit the customer's development platform. This information is useful during the initial development of the hardware ASIC and to get familiar with AMSS89x6, AMSS89x9, AMSS8952, AMSS8937, AMSS8953 and AMSS8976 software.

This customization guide is intended for software engineers who want to use or configure MSM8916/MSM8936/MSM8939/MSM8909/MSM8952/MSM8976/MSM8937/MSM8953 device GPIOs in the modem subsystem.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Overview

2.1 Hardware overview

This section provides an overview of the hardware aspects of the GPIOs. There are a total of

122 GPIOs in the MSM8916/MSM8936/MSM8939

112 GPIOs in MSM8909

134 GPIOs in MSM8937 and MSM8952

142 GPIOs in MSM8953

145 GPIOs in MSM8976

GPIO pad structure

Figure 2-1 shows a conceptual structure diagram of a GPIO pad.

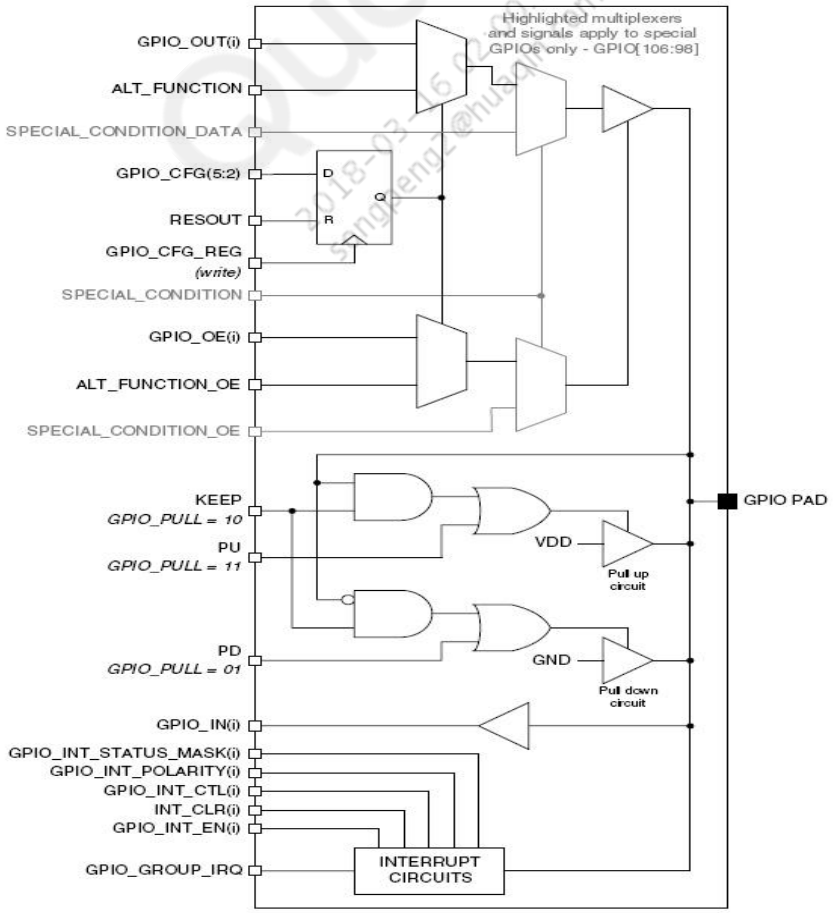


Figure 2-1 Conceptual structure diagram of GPIO pad

As shown, a GPIO pin can be configured as a general input, output, or one of several alternate functions. When used as an input, a GPIO pin can also be set up as an interrupt source, and a subset of GPIOs are capable of generating interrupts that can bring the MSM™ out of deep sleep (XO shutdown or VDD minimization). GPIOs can also be configured to have internal pull circuits (up, down, or no-pull) and different driving strengths.

2.1.1 GPIO registers

This section provides an overview of some important GPIO registers.

2.1.1.1 Configuration

The configuration of a GPIO is controlled by GPIO_CFGn, n = 0 ... 121 registers. Each register controls the configuration of one GPIO. Table 2-1 lists the bit field for the GPIO_CFGn register.

Table 2-1 GPIO_CFGn register bit field

Bits	Name	Description
10	GPIO_HIHYS_EN	Controls the hysteresis of GPIO[n] on input modes
9	GPIO_OE	Controls the OE for GPIO[n] when it is in GPIO mode
8:6	DRV_STRENGTH	Controls the GPIO pad drive strength; this applies regardless of the FUNC_SEL field selection ▪ 000 – 2 mA ▪ 001 – 4 mA ▪ 010 – 6 mA ▪ 011 – 8 mA ▪ 100 – 10 mA ▪ 101 – 12 mA ▪ 110 – 14 mA ▪ 111 – 16 mA
5:2	FUNC_SEL	Many of the GPIO pads have one or more functional hardware interfaces behind them. This field controls how the pad is used. Set this to the appropriate value for the function desired. It is GPIO mode when this field is set to 0.
1:0	GPIO_PULL	The pad can be configured to employ an internal weak pull-up, pull-down, keeper, or no-pull function; this applies regardless of the FUNC_SEL field selection. ▪ 00 – No-pull ▪ 01 – Pull-down (PD) ▪ 10 – Keeper ▪ 11 – Pull-up (PU)

2.1.1.2 Input and output

When the FUNC_SEL of a GPIO is set to 0, the GPIO is used as a general purpose pin, and it can be used as either an input or output. This is controlled by the GPIO_OE bit of the GPIO_CFGn register. When this bit is set to 1, it is an output pin. If the bit is set to 0, it is an input pin.

When set as an output, the GPIO's state is controlled by GPIO_IN_OUTn, n = 0 ... 121 registers. When set as an input, the GPIO's state can be read from the same register. Table 2-2 lists the bit field of the GPIO_IN_OUTn register.

Table 2-2 GPIO_IN_OUT register bit field

Bits	Name	Description
31:2	Reserved	Reserved field
1	GPIO_OUT	Controls the output state of an output pin
0	GPIO_IN	Allows the state of an input pin to be read

2.1.1.3 Interrupt

All GPIOs can be set up as interrupt sources. GPIO interrupt is controlled by the GPIO_INTR_CFGn, n = 0 ... 121 register. Table 2-3 lists the bit field of the GPIO_INTR_CFGn register.

Table 2-3 GPIO_INTR_CFGn bit field

Bit	Name	Description
31:9	Reserved	Reserved field
8	DIR_CONN_EN	Tells the Top Level Module Multiplex (TLMM) that GPIO[n] is being used as a direct connect interrupt. ▪ 0 – Disables the GPIO as a direct connect interrupt ▪ 1 – Enables the GPIO as a direct connect interrupt
7:5	TAEGET_PROC	Determines to which processor a summary interrupt from GPIO[n] gets routed ▪ 0x0: WCSS – Routes the GPIO[n] signal to the WCSS summary interrupt ▪ 0x1: Sensors – Routes the GPIO[n] signal to the sensors summary interrupt ▪ 0x2: LPA_DSP – Routes the GPIO[n] signal to the LPASS summary interrupt ▪ 0x3: RPM – Routes the GPIO[n] signal to the RPM summary interrupt ▪ 0x4: KPSS – Routes the GPIO[n] signal to the KPSS summary interrupt ▪ 0x5: MSS – Routes the GPIO[n] signal to the MSS summary interrupt ▪ 0x6: TZ – Routes the GPIO[n] signal to the TZ summary interrupt ▪ 0x7: None – Does not route to any processor subsystem; this is the default setting
4	INTR_RAW_STATUS_EN	Enables the RAW status for the summary interrupt on this GPIO; this is a power-saving mechanism and should be left disabled unless necessary ▪ 1 – Enable ▪ 0 – Disable
3:2	INTR_DECT_CTL	Controls the edge or level detection of the interrupt controller ▪ 0x0: LEVEL – Level sensitive ▪ 0x1: POS_EDGE – Positive edge sensitive ▪ 0x2: NEG_EDGE – Negative edge sensitive ▪ 0x3: DUAL_EDGE – Sensitive to both edges

Bit	Name	Description
1	INTR_POL_CTL	Controls the polarity detection of the interrupt controller ▪ Polarity 1 corresponds to active high ▪ Polarity 0 corresponds to active low ▫ 0 – POLARITY_0 ▫ 1 – POLATRITY_1
0	INTR_ENABLE	Controls the generation of summary interrupt ▪ 1 – Enable ▪ 0 – Disable

The interrupt's trigger type is controlled by INTR_POL_CTL and INTR_DETC_CTL. Setting/clearing the INTR_ENABLE field will enable/disable the corresponding GPIO interrupt.

The interrupt status for GPIO[n] is recorded in the GPIO_INTR_STATUSn, n = 0 ... 121 register. When read, this register returns the status of the corresponding GPIO's interrupt status.

- The interrupt is active when it returns 1. To clear the GPIO interrupt status, write a 0 into this register.
- The interrupt is not active when it returns 0. To set the interrupt, write a 1 into this register.

2.2 Software overview

The GPIO, or TLMM, driver is built on top of the Hardware Abstraction Layer (HAL) framework. This HAL software layer placed between the hardware and the driver code is designed to insulate the driver code from specific details of the underlying hardware for better portability and easier maintenance.

2.2.1 Code locations

The driver code referred to in this document can be found under the following paths:

- TLMM code – \core\systemdrivers\tlmm\
- GPIO interrupt driver code – \core\systemdrivers\GPIOInt
- Header files – \core\api\systemdrivers
- TLMMChipset.xml file – boot_images\core\systemdrivers\tlmm\config\msm8916

GPIO dump scripts – boot_images\core\systemdrivers\tlmm\scripts\<msmxxx>\tlmm_gpio_hw.cmm

3 GPIO HAL Layer

3.1 HAL and DAL introduction

Software development typically follows and lags behind hardware development. The process takes a long time because hardware and software design steps are sequential. This results in long time-to-market cycles.

To reduce the time it takes to provide an integrated solution, the software architecture framework in the MSM8916/MSM8936/MSM8939/MSM8909/MSM8952/MSM8937/MSM8976 and MSM8953 devices are designed to satisfy the following goals:

- Extract the hardware dependencies to decouple the driver from the underlying hardware design
- Enhance the software portability by encapsulating customer-configurable parameters
- Enable faster software integration and product launch by allowing the software development to begin before hardware finalization

In the new software architecture, HAL provides a layer that insulates the drivers from the details of the underlying hardware design. Moreover, a Device Abstraction Layer (DAL) is also placed between the driver code and application code to provide similar insulation between device drivers and application software.

Figure 3-1 shows the software architecture with the addition of the HAL and DAL.

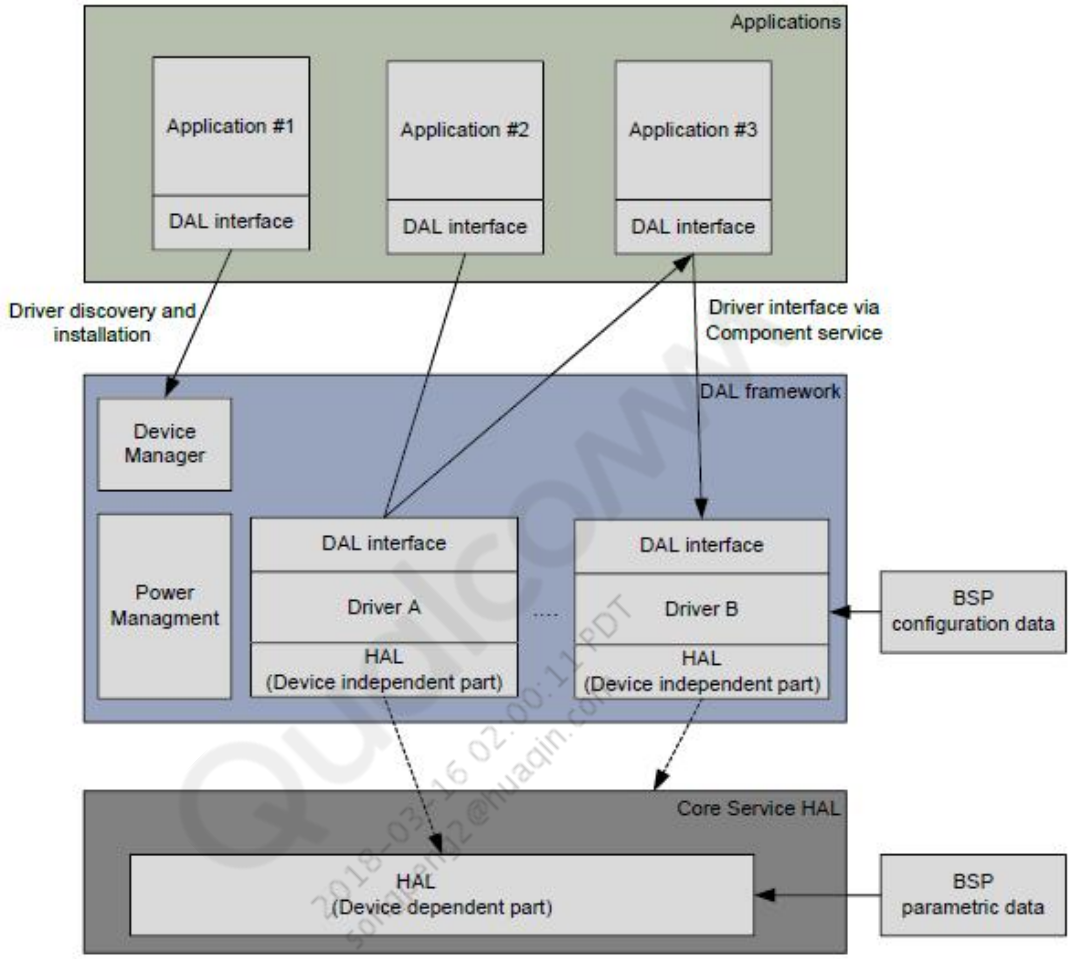


Figure 3-1 Software architecture with HAL and DAL

In this software architecture, the application software invokes the services of the device drivers through the DAL, and the device drivers access the hardware through the HAL. The HAL contains two parts, the device-dependent part, which is the actual implementation based on the underlying hardware, and the device-independent part, which provides the interface to device drivers on the top. The Board Support Package (BSP) provides parametric inputs, typically in a table, to configure the driver or hardware.

Characteristics of the HAL software architecture are:

- HAL APIs are immutable; note that the versioning scheme allows the extension of the APIs.
- Only device drivers are allowed to use HAL APIs.
- Target-specific configuration data is placed in BSP data.
- The HAL implementation is independent of the kernel.
- All HAL APIs are synchronous.
- Generally, HAL APIs include the common functions, initialization, reset, save, and restore.

For more information on QTI's HAL architecture, see *Presentation: Introduction to the Qualcomm Hardware Abstraction Layer (HAL)* (80-VH456-1).

3.2 TLMM interface of HAL

This section describes the HAL interface for the TLMM.

3.2.1 Type definitions

The HAL TLMM basic type definitions related to GPIO properties are:

```
typedef enum
{
  HAL_TLMM_NO_PULL   = 0,
  HAL_TLMM_PULL_DOWN = 1,
  HAL_TLMM_KEEPER    = 2,
  HAL_TLMM_PULL_UP   = 3
}HAL_tlmm_PullType;

typedef enum
{
  HAL_TLMM_INPUT  = 0,
  HAL_TLMM_OUTPUT = 1,
}HAL_tlmm_DirType;

typedef enum
{
  HAL_TLMM_DRIVE_2MA  = 0,
  HAL_TLMM_DRIVE_4MA  = 1,
  HAL_TLMM_DRIVE_6MA  = 2,
  HAL_TLMM_DRIVE_8MA  = 3,
  HAL_TLMM_DRIVE_10MA = 4,
  HAL_TLMM_DRIVE_12MA = 5,
  HAL_TLMM_DRIVE_14MA = 6,
  HAL_TLMM_DRIVE_16MA = 7
}HAL_tlmm_DriveType;


/*
 * The HAL_tlmm_GpioType is the basic GPIO configuration element.  It
 * contains information about the configuration of a particular GPIO.
 */
typedef struct
{
  uint8 nFunc;        /* Function select for a GPIO (0-15). */
```

```
  uint8 nDir;         /* Direction select for a GPIO (INPUT/OUTPUT). */
  uint8 nPull;        /* Pull (PULL-DOWN, PULL-UP, KEEPER, NO-PULL */
  uint8 nDrive;       /* Drive strength (2-16 mA in even increments). */
}HAL_tlmm_GpioType;
```

These define the different direction, pull, and driving strength types of a GPIO's configuration.

3.2.2 Macros

Some HAL macros are visible to the TLMM driver. These macros extract the properties of the GPIO configuration.

- HAL_GPIO_NUMBER – Returns the GPIO number based on configuration data
- HAL_OUTVAL – Extracts the output state from configuration data
- HAL_DRVSTR_VAL – Returns the driving strength
- HAL_PULL_VAL – Returns the pull direction
- HAL_DIR_VAL – Returns the direction information, either input or output
- HAL_FUNC_VAL – Returns the function value, either 0 for general purpose, or nonzero for an alternate function

3.2.3 Functions

This section describes some HAL TLMM APIs. This list of functions is to help readers understand the GPIO driver that is described in the next chapter. The use of these functions should be limited to the GPIO driver only. Application software should not call these functions directly.

3.2.3.1 Initialization

This function initializes the hardware buffers and address structures, and returns the pointer to the version of this HAL module via the input parameter.

```
void HAL_tlmm_Init ( char ** ppszVersion );
```

3.2.3.2 Configure GPIOs

This function configures the specified GPIO.

```
void HAL_tlmm_ConfigGpio(uint32 nWhichConfig);
```

Other functions that involve the GPIO configuration include the function to configure a group of GPIOs, the function to configure GPIOs for power-collapse, the function to restore GPIO configuration after power-collapse, and the function to configure keysense GPIOs, etc. These functions are listed as remove power-collapse configuration and keysense configuration.

```
void HAL_tlmm_ConfigGpioGroup(const uint32 nWhichGpioSet[],
                              uint16 nWhatSize );
```

The following API obtains the current GPIO configuration from the GPIO_CFGn register.

```
void HAL_tlmm_GetConfig( uint32 nGpioNumber,
                         HAL_tlmm_GpioType* tGpio );
```

3.2.3.3 Read and write

The following function reads values from the GPIO's GPIO_IN registers. A single GPIO is read each time this function is called, and the value is returned as a Boolean type, where TRUE indicates High state and FALSE indicates Low state.

```
boolean HAL_tlmm_ReadGpio     ( uint32 nWhichConfig );
```

To write a value to the GPIO pin, i.e., push a value to the GPIO_OUT register, the following function is called. The value to be written is stored in a Boolean-type parameter.

```
void HAL_tlmm_WriteGpio       ( uint32 nWhichConfig, boolean bValue );
```

In the TLMM driver, the GPIOs are divided into different groups, i.e., GPIO_GROUP_AUDIO_PCM, GPIO_GROUP_USB, GPIO_GROUP_SDC1, etc. Sometimes it is necessary to write to a group of GPIOs at once. The following HAL API is constructed for this purpose.

```
void HAL_tlmm_WriteGpioGroup(const uint32 nWhichConfigSet[],
                             uint16 nWhatSize,
                             boolean bWriteVal);
```

The following API reads the output value of an output GPIO pin from its GPIO_IN_OUTn register.

```
boolean HAL_tlmm_GetOutput( uint32 nWhichGpio );
```

3.2.3.4 Set port

The set port API of the TLMM sets certain TLMM registers so that the associated GPIOs are set to a particular special function, e.g., it can set UIM1 and UIM2 pad control.

```
void HAL_tlmm_SetPort(HAL_tlmm_PortType ePort, uint32 mask,
                      uint32 value);
```

3.3 GPIO interrupt interface of HAL

The HAL code for a GPIO interrupt is listed under \core\systemdrivers\hal\gpioint\.

3.3.1 Basic definition

The following definition enumerates the five possible trigger types of GPIO interrupts.

```
typedef enum
{
  HAL_GPIOINT_TRIGGER_HIGH     = 0,
  HAL_GPIOINT_TRIGGER_LOW      = 1,
  HAL_GPIOINT_TRIGGER_RISING   = 2,
  HAL_GPIOINT_TRIGGER_FALLING  = 3,
  HAL_GPIOINT_TRIGGER_DUAL_EDGE  = 4
} HAL_gpioint_TriggerType;
```


3.3.2 Functions

This section describes some HAL GPIO interrupt APIs. This list of functions is to help readers understand the GPIO interrupt driver code. The use of these functions should be limited to the TLMM driver only. Application software should not call these functions directly.

3.3.2.1 Initialization

The following function initializes the HAL's GPIO interrupt controller.

```
void HAL_gpioint_Init (char **ppszVersion)
```

3.3.2.2 Restore and save

The following function saves the current hardware context.

```
void HAL_gpioint_Save (void);
```

The following function restores the hardware context saved by the HAL_gpioint_Save function.

```
void HAL_gpioint_Restore (void);
```

3.3.2.3 Enable and disable

This function enables (unmasks) the given interrupt.

```
void HAL_gpioint_Enable (uint32 nGPIO);
```

This function disables (masks) the given interrupt.

```
void HAL_gpioint_Disable ( uint32 nGPIO);
```

3.3.2.4 Set trigger

This function sets the trigger type for the given interrupt.

```
void HAL_gpioint_SetTrigger (uint32 nGPIO,
                             HAL_gpioint_TriggerType eTrigger);
```

3.3.2.5 Clear

This function clears the given interrupt.

```
void HAL_gpioint_Clear (uint32 nGPIO);
```

3.3.2.6 Inquiry

Some HAL functions respond to certain inquiries of the driver code. The following function is used to retrieve the next pending GPIO interrupt in the given group. This function returns a GPIO number if the GPIO interrupt is pending or HAL_GPIOINT_NONE if no GPIO interrupt is pending.

```
void HAL_gpioint_GetPending (HAL_gpioint_GroupType eGroup,
                             uint32 *pnGPIO);
```

This function returns the number of supported GPIO interrupts on this platform.

```
void HAL_gpioint_GetNumber (uint32 *pnNumber);
```

This function retrieves the current trigger type for the given interrupt.

```
void HAL_gpioint_GetTrigger (uint32 nGPIO,
                             HAL_gpioint_TriggerType *peTrigger);
```

This function returns if the given interrupt is supported on this platform.

```
boolean HAL_gpioint_IsSupported (uint32 nGPIO);
```

This function checks if a given interrupt is waiting to be serviced.

```
boolean HAL_gpioint_IsPending (uint32 nGPIO);
```

This function returns if the given interrupt is enabled.

```
boolean HAL_gpioint_IsEnabled (uint32 nGPIO);
```

4 GPIO DAL Driver

This chapter describes the API for the DAL Top-Level Mode Multiplexer (DALTLMM). The DAL provides an interface to the TLMM driver. All drivers can use the Device Driver Interface (DDI) provided by the TLMM to talk to the TLMM driver. The driver internally uses the TLMM HAL API to perform the necessary hardware functions. Therefore, the HAL is never exposed to the client's use of the TLMM driver. These APIs provide a uniform and efficient interface to the driver independent of the operating system and the processor on which the driver resides, which allows quick integration of the ASICs.

The TLMM driver provides an interface for other driver layer modules to interact with the TLMM hardware block. This is the TLMM DAL interface. The TLMM DAL APIs are platform- and operating-system independent.

4.1 GPIO configurations

A GPIO generally has two types of configurations, run-time and sleep.

4.1.1 Run-time configuration

The MSM8916/MSM8936/MSM8939/MSM8909/MSM8952/MSM8937/MSM8976 and MSM8953 devices do not rely on the TLMM driver to initialize the GPIOs run-time configuration. The software module that owns the GPIO configures it into the appropriate state.

The xml files in the following directory define GPIO IDs and the alternate function for each GPIO:

```
\modem_proc\core\systemdrivers\tlmm\config\msm89xx\
```

Each ID starts with a string name and corresponds to an appropriate GPIO and its alternate function. For example, the following line defines string "uim2_data" to be associated with GPIO 0's alternate function 1.

```
<props name="uim2_data" type="TLMMGpioIdType">{0, 1}</props>
```

These IDs are used when a driver requests and configures GPIOs with API DalTlmm_ConfigGpioId (see Section 4.4.2). When this API is used, the GPIO driver locks the GPIO in software and prevents other modules from accessing the GPIO without the correct GPIO ID.

It does not save the current GPIO configuration data in software, since each GPIO has its own configuration register. Current GPIO configuration information is obtained by reading the GPIO register directly.

You can run tlmm_gpio_hw.cmm script from RPM t32 to get the run-time configuration of all the GPIOs.

4.1.2 GPIO sleep configuration

The sleep configuration of a GPIO is the configuration applied to a GPIO when the device enters deep sleep. It is dependent on the peripheral that the device is connected to and the peripheral's state during the MSM sleep. Some guidelines on deciding the sleep configuration are:

- When the MSM goes into sleep, it is desirable to avoid the following two scenarios to avoid leakage current on a GPIO pad:
  - Conflict between the MSM and external device – A GPIO should be driven by either the MSM or external device.
  - Floating pin – A GPIO should have a definite state during sleep. If a GPIO is left floating around Vdd/2, the quiescent current in the IC circuit is the highest.
- Avoid these scenarios by examining how the GPIOs are used and how peripheral devices behave during sleep. If the external device drives the GPIO during sleep, the GPIO should be set to input with the desirable pull direction. If the external device does not drive the pin during sleep, the GPIO can be set to output with the appropriate logic state.
- It is important to distinguish the GPIO's logic state and its internal pull. The internal pull is relatively weak, around 100 kΩ, and it determines the GPIO state only when neither the MSM nor external device drives the pin.
- Do not expect a GPIO to have a High state during sleep when PULL_HIGH is defined for the pin.

The sleep configuration of GPIOs is saved in TLMMChipset.xml (boot_images\core\systemdrivers\tlmm\config\<chipset>\TLMMChipset.xml). The settings in this file are loaded into a TLMM data structure and applied during boot.

You can run tlmm_gpio_sleep.cmm script from RPM t32 to get the sleep configuration of all the GPIOs stored in the TLMM data structure.

4.1.3 GPIO configuration timeline

A GPIO goes through different configurations during MSM operation as follows:

- Once the MSM boots up, the GPIOs are initialized in their default hardware state. Then the TLMM driver initializes the GPIOs with their sleep configuration if DALTLMM_PRG_YES is specified for the GPIO in TLMMChipset.xml. This ensures that any unused GPIOs remain in a low power state once the MSM starts up. If it is desirable to avoid this type of initial configuration on a GPIO, DALTLMM_PRG_NO can be specified for the GPIO in TLMMChipset.xml. This process takes place during SBL inside API DALTLMMState_Init().
- Then the initialization code of the software module that owns the GPIO needs to carry out the GPIO initialization and configure its GPIO to its active configuration. During its operation, it is possible that a GPIO's configuration may be changed, e.g., from an output pin to an input pin.
- When the software module that owns the GPIO goes into sleep, it needs to apply the sleep configuration on the GPIO to conserve power. This can be done by calling a TLMM API, e.g., DalTlmm_ConfigGpio, with parameter DAL_TLMM_GPIO_DISABLE or DalTlMM_ConfigGpioIdInactive with GPIO ID. The TLMM automatically applies the sleep configuration stored in the internal variable gptState, whose settings are loaded during boot from TLMMChipset.xml. Once the software module comes out of suspend, it needs to restore the GPIOs back to their run-time configuration by calling the appropriate APIs.

4.1.4 Freeze IO

NOTE: Numerous changes were made in this section.

When MSM enters VDD minimization (VDD MIN), the MSM Power Manager (MPM) decides whether to enable freeze IO based on the `Freeze IOs` setting in the following MPM_BSP_DATA variable:

```
File: \rpm_proc\core\power\mpm\hal\bsp\source\9x07\BSPmpm.c.
/*
 * Target specific MPM Hardware configuration.
 */
BSP_mpm_ConfigDataType MPM_BSP_DATA =
{
  /* MPM Configuration */
  {
    /* Wakeup Delays */
    {
      xoDelays, deepSleepExitDelay
    },

    /* IO Cfg */
    {
      /* Freeze  Clamp  SW Ebi1    Warm Boot    Warm Boot
       * IOs     IOs    Ctl Enable Freeze EBI1  Freeze EBI2 */
      TRUE,      TRUE,  TRUE,      TRUE,        TRUE
    },
```

When freeze IO is enabled, i.e. `Freeze IOs` is set to `TRUE` as in the above code, the internal PULL of all GPIOs becomes KEEPER when the MSM enters VDD minimization, regardless of whether the PULL is set to PULL_DOWN, PULL_UP, or NO_PULL before sleep.

The KEEPER setting tries to maintain the previous state of the GPIO. The purpose of this design is to eliminate the possibility of conflict between internal PULL and external PULL/drive during VDD MIN and reduce leakage current. However, it sometimes has unintended effects on customer designs. The following example illustrates the difference between KEEPER and normal PULL.

The GPIO is configured as INPUT, PULL_UP right before VDD minimization with freeze IO enabled. When the MSM enters VDD MIN, the GPIO is connected to GND, and its state is LOW despite the internal PULL_UP. Once MSM is in VDD MIN, the GPIO is disconnected from GND and connected to an external PULL_UP resistor of ~100 kΩ. However, the GPIO voltage does not return fully to VDD (1.8 V). Instead, it may settle to anywhere between 0.4 V to 1.2 V and not be considered a digital HIGH signal.

This settling is because the GPIO internal PULL changes to KEEPER upon entering VDD MIN when freeze IO is enabled. Since the GPIO is at a LOW state upon entering sleep, the KEEPER tries to maintain the LOW state and becomes effectively PULL_DOWN. Even when the GPIO is switched from GND to PULL_UP externally, the internal PULL_DOWN contends with the external PULL_UP resistor to keep the GPIO floating around VDD/2, which is a digitally ambiguous state.

In this scenario, to ensure that the GPIO reaches a digitally HIGH state, a smaller PULL_UP resistor (~10 kΩ) must be used to overcome the internal KEEPER and bring the GPIO to a fully HIGH state during VDD MIN.

4.2 DALTLMM data definitions

This section describes public macros and data structures used by the TLMM module.

4.2.1 DAL_GPIO_CFG macros

The DAL_GPIO_CFG macro is used to generate a GPIO's configuration signal (DALGpioSignalType).

```
#define DAL_GPIO_CFG(gpio, func, dir, pull, drive) \
         (((gpio) & 0x3FF)<< 4 | \
          ((func) & 0xF)|        \
          ((dir)  & 0x1) << 14|  \
          ((pull) & 0x3) << 15|  \
          ((drive)& 0xF) << 17| DAL_GPIO_VERSION)
```

Parameters are:

- gpio – The GPIO number associated with this signal
- func – The GPIO function associated with this signal, 0 for general purpose, nonzero for alternate functions
- dir – The direction when the GPIO is set as a general purpose pin
- pull – The pull type
- drive – The drive strength for this signal

For example, the following function generates the configuration data for a GPIO 0 general purpose output.

```
uint32 gpio0_cfg = DAL_GPIO_CFG(0, 0, DAL_GPIO_OUTPUT, \
                 DAL_GPIO_NO_PULL, DAL_GPIO_2MA);
```

The following macro allows the user to specify an additional outval that defines the Output state of the GPIO.

```
#define DAL_GPIO_CFG_OUT(gpio, func, dir, pull, drive, outval) \
          (DAL_GPIO_CFG((gpio), (func), (dir), (pull), (drive))\
          | (((outval)|0x2) << 0x15))
```

This is necessary when defining the low power configuration of an output GPIO.

4.2.2 GPIO state variable

The following variable is the main state structure containing all configuration and attribute information pertaining to the set of GPIOs on a particular target.

```
DALTLMMStateType* pgtState;
```

It contains the sleep configuration data that is loaded from TLMMChipset.xml and indicates whether a GPIO is currently active. The TLMM driver does not save GPIO active configurations, since each GPIO has its own configuration register in the MSM8916 device.

4.3 DALTLMM interface description

NOTE: Numerous changes were made in this section.

The DALTLMM APIs provide functionalities for GPIO configuration, read, write, etc. The implementation of these APIs is in DALTLMM.c.

4.3.1 DalTlmm_GetGpioId

This API retrieves a GPIO identifier corresponding to an input string.

```
DALResult DalTlmm_GetGpioId(DalDeviceHandle * _h, const char* pszGpio,
DALGpioIdType* pnGpioId);
```

The string name must match exactly the expected name for the hardware functionality as defined in the xml file described in Section 4.2.1. If the API returns successfully, the retrieved ID is associated with the alternate function as defined in the xml file.

4.3.2 DalTlmm_ConfigGpioId

When this function is called to configure the GPIO, the nGpioId passed in (which is requested through DalTlmm_GetGpioId) is checked to see whether it matches the internal ID stored in the data structure ganGpioIdUsers. If yes, it configures the GPIO. If no, it returns DAL_ERROR.

```
DALResult DalTlmm_ConfigGpioId(DalDeviceHandle * _h, DALGpioIdType nGpioId,
DalTlmm_GpioConfigIdType* pUserSettings);
```

4.3.3 DalTlMM_ConfigGpioIdInactive

This function sets the GPIO to its sleep configuration based on the GPIO ID passed in. The sleep configuration defined in the TLMMChipset.xml file of the boot image is used.

```
DALResult DalTlmm_ConfigGpioIdInactive(DalDeviceHandle * _h, DALGpioIdType
nGpioId);
```

This function also checks the nGpioId to see whether it matches the ID stored internally. If yes, it configures the GPIO. If no, it return DAL_ERROR.

4.3.4 DalTlmm_SelectGpioIdMode

This API sets a GPIO to a general purpose IO pin or alternate function based on the eMode parameter. It also checks nGpioId to verify that it matches the GPIO ID that is stored in the xml file described in Section 4.2.1.

```
DALResult DalTlmm_SelectGpioIdMode( DalDeviceHandle * _h, DALGpioIdType
nGpioId, DalGpioModeType eMode, DalTlmm_GpioConfigIdType* pUserSettings);
```

If nGpioId does not match the internal GPIO ID, it returns DAL_ERROR. If eMode is TRUE, the GPIO is set as a general purpose IO pin. If eMode is FALSE, the GPIO is set to its alternate function based on nGpioId.

4.3.5 DalTlmm_GpioIdIn

This function checks the nGpioId passed in to see whether it matches the ID stored in the internal data structure. If no, it returns DAL_ERROR. If yes, it reads out the GPIO state.

```
DALResult DalTlmm_GpioIdIn(DalDeviceHandle * _h, DALGpioIdType nGpioId,
DALGpioValueType *eValue);
```

4.3.6 DalTlmm_GpioIdOut

This function checks the nGpioId passed in to see whether it matches the ID stored in the internal data structure. If no, it returns DAL_ERROR. If yes, it sets the GPIO state.

```
DALResult DalTlmm_GpioIdOut(DalDeviceHandle * _h, DALGpioIdType nGpioId,
DALGpioValueType eValue);
```

A GPIO must be configured as GP OUTPUT for this function to take effect.

4.3.7 DalTlmm_GetInactiveConfig

This API returns the sleep configuration of a GPIO:

```
DALResult DalTlmm_GetInactiveConfig
(
  DalDeviceHandle  *_h,
  uint32           gpio_number,
  DALGpioSignalType *gpio_config
);
```

The following API can be used to set the inactive configuration of a GPIO:

```
DALResult DalTlmm_SetInactiveConfig
(
  DalDeviceHandle  *_h,
  uint32           gpio_number,
  DALGpioSignalType gpio_config
);
```

The low-power configuration parameter should use the macro DAL_GPIO_CFG_OUT to specify the output state if the GPIO is set to an output pin during sleep.

4.3.8 DalTlmm_ConfigGpio

This function is used to configure a single GPIO based on the parameters passed in. It requires a DAL device handle. If the enable value is DAL_TLMM_GPIO_DISABLE, then the sleep configuration of the GPIO is applied instead of the configuration passed in.

```
DALResult DalTlmm_ConfigGpio(DALDeviceHandle* _h,
                             DALGpioSignalType gpio_config,
                             DALGpioEnableType enable);
```

The following is an example of the DalTlmm API configuring the GPIO 0 as a pull-up input pin and disabling it after usage.

```
DalDeviceHandle *htlmm;
DALResult result;
uint32 gpio0_cfg = DAL_GPIO_CFG(0, 0, DAL_GPIO_INPUT, \
                   DAL_GPIO_PULL_UP, DAL_GPIO_2MA);

// Create a TLMM handle
result = DAL_DeviceAttach(DALDEVICEID_TLMM, &htlmm);
if (result != DAL_SUCCESS) { goto error; }

result = DalDevice_Open(htlmm, DAL_OPEN_SHARED);
if (result != DAL_SUCCESS) { goto error; }

// Enable GPIO0
result = DalTlmm_ConfigGpio(htlmm, gpio0_sig, DAL_TLMM_GPIO_ENABLE);
if (result != DAL_SUCCESS) { goto error; }

// GPIO operation
...
```

```
// Disable GPIO0
result = DalTlmm_ConfigGpio(htlmm, gpio0_sig, DAL_TLMM_GPIO_DISABLE);
if (result != DAL_SUCCESS) { goto error; }

// Remove the handle
DalDevice_Close(htlmm);
DAL_DeviceDetach(htlmm);
```

NOTE: It is necessary to create a TLMM device handle before calling any DalTlmm APIs, and disabling a GPIO causes the TLMM driver to apply the sleep configuration on the GPIO.

4.3.9 DalTlmm_ConfigGpioGroup

This function configures a group of GPIOs passed in as a pointer to an array of configurations. It returns a DAL_ERROR result when the group value passed in is not the correct value. Depending on the enable flag enumeration, the GPIO group is either configured to its active configuration or its sleep configuration.

```
DALResult DalTlmm_ConfigGpioGroup
(
  DalDeviceHandle * _h,
  DALGpioEnableType  enable,
  DALGpioSignalType* gpio_group,
  uint32             size
);
```


4.3.10 DalTlmm_GetCurrentConfig

This API returns the current configuration of the GPIO. It reads the configuration from the hardware register and converts it into a DALGpioSignalType using the DAL_GPIO_CFG macro.

```
DALResult DalTlmm_GetCurrentConfig
(
  DalDeviceHandle  *_h,
  uint32           gpio_number,
  DALGpioSignalType *gpio_config
);
```

4.3.11 DalTlmm_GetGpioNumber

This function returns the GPIO number based on the GPIO configuration passed in.

```
DALResult DalTlmm_GetGpioNumber
(
  DalDeviceHandle   *_h,
  DALGpioSignalType gpio_config,
  uint32            *gpio_number
);
```

4.3.12 DalTlmm_GetGpioStatus

This API returns the current status of a GPIO.

```
DALResult DalTlmm_GetGpioStatus
(
  DalDeviceHandle   *_h,
  uint32            gpio_number,
  DALGpioStatusType *status
);
```

4.3.13 DalTlmm_GetInactiveConfig

This API returns the sleep configuration of a GPIO.

```
DALResult DalTlmm_GetInactiveConfig
(
  DalDeviceHandle  *_h,
  uint32           gpio_number,
  DALGpioSignalType *gpio_config
);
```

This API sets the inactive configuration of a GPIO.

```
DALResult DalTlmm_GetInactiveConfig
(
  DalDeviceHandle  *_h,
  uint32           gpio_number,
  DALGpioSignalType *gpio_config
);
```

The low power configuration parameter uses the macro DAL_GPIO_CFG_OUT to specify the Output state if the GPIO is set to an output pin during sleep.

4.3.14 DalTlmm_SetPort

This API sets specific TLMM ports, e.g., UIM, for certain GPIOs.

```
DALResult DalTlmm_SetPort
(
  DalDeviceHandle *_h,
  DALGpioPortType port,
  uint32          value
)
```

4.3.15 DalTlmm_GpioIn

This function returns the value of DAL_GPIO_HIGH_VALUE or DAL_GPIO_LOW_VALUE corresponding to the input value of the GPIO.

```
DALResult DalTlmm_GpioIn
(
  DalDeviceHandle *_h,
  DALGpioSignalType gpio_config,
  DALGpioValueType*value
);
```

4.3.16 DalTlmm_GpioOut

This function drives an output GPIO pin to the High or Low state. Acceptable values are DAL_GPIO_HIGH_VALUE and DAL_GPIO_LOW_VALUE. The values will not take effect if the ownership is not configured correctly.

```
DALResult DalTlmm_GpioOut
(
  DalDeviceHandle  *_h,
  DALGpioSignalType gpio_config,
  DALGpioValueType  value
);
```

The following API sets a group of output GPIOs to the same state at one time.

```
DALResult DalTlmm_GpioOutGroup
(
  DalDeviceHandle   *_h,
  DALGpioSignalType *gpio_group,
  uint32            size,
  DALGpioValueType  value
);
```

5 GPIO Interrupt Driver

Each GPIO can act as an interrupt source. The GPIO interrupt driver allows configuration of a GPIO interrupt's trigger type and enabling/disabling the GPIO interrupt.

5.1 Data definitions

5.1.1 Trigger and ISR types

The triggering type of an interrupt is defined as TRIGGER_HIGH and TRIGGER_LOW and are level triggers with the corresponding polarity. TRIGGER_RISING, TRIGGERING_FALLING, and TRIGGER_DUAL_EDGE are edge trigger types.

```
typedef enum{
GPIOINT_TRIGGER_HIGH,
GPIOINT_TRIGGER_LOW,
GPIOINT_TRIGGER_RISING,
GPIOINT_TRIGGER_FALLING,
GPIOINT_TRIGGER_DUAL_EDGE,
PLACEHOLDER_GPIOIntTriggerType = 0x7fffffff
}GPIOIntTriggerType;
```

The following defines a pointer to an ISR function that takes a given parameter as an argument, typically the IRQ number:

```
typedef void * (*GPIOINTISR)(GPIOINTISRCtx);
```

These types are used when calling DAL GPIO interrupt APIs.

5.1.2 DAL GPIO interrupt data structure

The following data structure contains debug information for GPIO interrupts.

```
/*
 * This is static GPIOInt state data. It can be accessed for debugging
 * GPIOInterrupts to see what is the current registration state of the
 * GPIO.
 */
```

```
static GPIOIntCntrlType GPIOIntData;
```

The following gpioint_cntrl data structure contains various GPIO interrupt information.

```
/*
 * GPIOIntCntrlType
 *
 * Container for all local data.
 *
 * initialized: Indicates if the driver has been started or not.
 *              Needed mostly because some compilers complain about
 *              empty structs.
 * table:       Table of registered GPIO_INT handler functions.
 * wakeup_isr:  ISR to invoke when a monitored GPIO interrupt triggers.
 * log:         Log storage.
 */
typedef struct
{
  /* GPIOInt Dev state can be added by developers here */

  /* Flag to Initialize GPIOInt_Init is called first
   * before anything else can attach
   */
  uint8                              GPIOInt_Init;

  /* interrupt_state Table of registered GPIO_INT handler functions */
  GPIOIntDataType                state[MAX_NUMBER_OF_GPIOS];

  /* Interrupt Log storage.*/
  GPIOIntLogType                 log;

  /* total number of GPIOs present on the target */
  uint32                         gpio_number;

  /*
   * Number of direct connect interrupts.
   */
  uint32                         direct_intr_number;

  /*
   * This keeps track of the gpios that are configured as
   * direct connect interrupts.
   */
  HAL_gpioint_ProcessorType      processor;
```

```
    DALSYSEventHandle              summary_intr_event;
    DALSYSEventHandle              default_event;
    uint32                         default_param;
    uint32                         summary_param;
    uint32                         summary_intr_id;
    #ifdef GPIOINT_USE_NPA
    npa_client_handle     npa_client;
    uint32                non_mpm_interrupts;
    #endif /* GPIOINT_USE_NPA*/
    GPIOINTISR                     wakeup_isr;

    /* Configuration map for direct connect interrupts.*/
    GPIOIntConfigMapType *gpioint_config_map;

    /*
     * fake trigger flag for gpios that are triggered in software.
     */
    uint32*                        gpioint_physical_address;
    uint32*                        gpioint_virtual_address;

    /*
     * The main interrupt controller that GPIOInt connects to.
     */
    uint8                          interrupt_controller;

} GPIOIntCntrlType;
```

NOTE: The most useful information for debugging is in the state and logs.

5.2 DAL GPIO interrupt APIs

This section describes some DAL GPIO interrupt APIs.

5.2.1 GPIO interrupt register and reregister

This API allows the user to register a GPIO interrupt ISR, specifying its triggering type. As a result of calling this API, the GPIO interrupt is enabled.

```
DALResult GPIOInt_RegisterIsr(DalDeviceHandle * _h,
              uint32 gpio, GPIOIntTriggerType trigger,
              GPIOINTISR isr,GPIOINTISRCtx param)
```

The following API deregisters and disables a GPIO interrupt.

```
DALResult GPIOInt_DeregisterIsr(DalDeviceHandle * _h,
              uint32  gpio, GPIOINTISR  isr)
```

5.2.2 GPO interrupt trigger

The following API sets the desired trigger type for a GPIO interrupt without providing an ISR.

```
DALResult GPIOInt_SetTrigger(DalDeviceHandle * _h,
              uint32  gpio, GPIOIntTriggerType  trigger)
```

5.2.3 IsInterruptEnabled

The state returns whether the GPIO interrupt has been enabled.

```
DALResult GPIOInt_IsInterruptEnabled(DalDeviceHandle * _h,
              uint32  gpio, uint32*  state)
```

5.2.4 IsInterruptPending

The state returns whether the GPIO interrupt is pending, i.e., enabled and triggered.

```
DALResult GPIOInt_IsInterruptPending(DalDeviceHandle * _h,
              uint32  gpio, uint32 *  state)
```

5.2.5 Example code

The following is an example of registering an ISR and changing the interrupt trigger.

```
DalDeviceHandle * hGPIOInt;
// Attach to DAL
if((DAL_DeviceAttach(DALDEVICEID_GPIOINT, &hGPIOInt)
!= DAL_SUCCESS) || (hGPIOInt == NULL) )
{
  // Clients need to handle a failure here.
}
// Register the ISR to GPIO 5 IRQ.
GPIOInt_RegisterIsr(hGPIOInt, 5, GPIOINT_TRIGGER_HIGH,
gpio5_isr, gpio5_param);
// Change the trigger to RISING
GPIOInt_SetTrigger(hGPIOInt, 5, GPIOINT_TRIGGER_RISING);
```

5.3 Wake-up interrupts

NOTE: Numerous changes were made in this section.

Only a subset of GPIOs is capable of waking up an MSM from deep sleep, i.e., XO shutdown or VDD minimization. When a GPIO that belongs to this subgroup of wake-up interrupts is enabled as an interrupt source, the GPIO interrupt driver recognizes this and forwards this to the RPM so that this interrupt can be enabled in the MPM when the RPM executes a deep sleep.

This process of identifying and registering a wake-up interrupt is transparent to the user code in the modem subsystem, and is done automatically by the GPIO interrupt driver. There is no additional configuration except that the user should be aware of which GPIOs can act as wake-up sources.

This information can be found in the following definition in RPM image.

```
File: \rpm_proc\core\power\mpm\hal\source\msm89xx\HALmpmintTable.c

HAL_mpmint_PlatformIntType aInterruptTable[HAL_MPMINT_NUM];
```

Table 5-1 and Table 5-2 list the wake-up capable GPIOs of MSM8916/MSM8936/MSM8939.

Table 5-1 TLMM_MPM_WAKEUP_INT_EN_0 bit field

Bit	Name	Description
31	GPIO_115	0: DISABLE
		1: ENABLE
30	GPIO_114	0: DISABLE
		1: ENABLE
29	GPIO_113	0: DISABLE

Bit	Name	Description
		1: ENABLE
28	GPIO_112	0: DISABLE
		1: ENABLE
27	GPIO_111	0: DISABLE
		1: ENABLE
26	GPIO_110	0: DISABLE
		1: ENABLE
25	GPIO_109	0: DISABLE
		1: ENABLE
24	GPIO_108	0: DISABLE
		1: ENABLE
23	GPIO_107	0: DISABLE
		1: ENABLE
22	GPIO_98	0: DISABLE
		1: ENABLE
21	GPIO_97	0: DISABLE
		1: ENABLE
20	GPIO_69	0: DISABLE
		1: ENABLE
19	GPIO_62	0: DISABLE
		1: ENABLE
18	GPIO_54	0: DISABLE
		1: ENABLE
17	GPIO_52	0: DISABLE
		1: ENABLE
16	GPIO_51	0: DISABLE
		1: ENABLE
15	GPIO_50	0: DISABLE
		1: ENABLE
14	GPIO_49	0: DISABLE
		1: ENABLE
13	GPIO_38	0: DISABLE
		1: ENABLE
12	GPIO_37	0: DISABLE
		1: ENABLE
11	GPIO_36	0: DISABLE
		1: ENABLE
10	GPIO_35	0: DISABLE
		1: ENABLE
9	GPIO_34	0: DISABLE
		1: ENABLE
8	GPIO_31	0: DISABLE

Bit	Name	Description
		1: ENABLE
7	GPIO_28	0: DISABLE
		1: ENABLE
6	GPIO_25	0: DISABLE
		1: ENABLE
5	GPIO_21	0: DISABLE
		1: ENABLE
4	GPIO_20	0: DISABLE
		1: ENABLE
3	GPIO_13	0: DISABLE
		1: ENABLE
2	GPIO_12	0: DISABLE
		1: ENABLE
1	GPIO_5	0: DISABLE
		1: ENABLE
0	GPIO_1	0: DISABLE
		1: ENABLE

Table 5-2 TLMM_MPM_WAKEUP_INT_EN_1 bit field

Bit	Name	Description
12	GPIO_68	0: DISABLE
		1: ENABLE
11	GPIO_66	0: DISABLE
		1: ENABLE
10	SDC2_DATA_3	0: DISABLE
		1: ENABLE
9	SDC2_DATA_1	0: DISABLE
		1: ENABLE
8	SDC1_DATA_3	0: DISABLE
		1: ENABLE
7	SDC1_DATA_1	0: DISABLE
		1: ENABLE
6	SRST_N	0: DISABLE
		1: ENABLE
5	GPIO_121	0: DISABLE
		1: ENABLE
4	GPIO_120	0: DISABLE
		1: ENABLE
3	GPIO_9	0: DISABLE
		1: ENABLE
2	GPIO_118	0: DISABLE

Bit	Name	Description
		1: ENABLE
1	GPIO_117	0: DISABLE
		1: ENABLE
0	GPIO_11	0: DISABLE
		1: ENABLE

A References

A.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Presentation: Introduction to the Qualcomm Hardware Abstraction Layer (HAL)	80-VH456-1

A.2 Acronyms and terms

Acronym or term	Definition
DAL	Device Abstraction Layer
DALTLMM	DAL Top-Level Mode Multiplexer
DDI	Device Driver Interface
GPIO	General Purpose Input Output
HAL	Hardware Abstraction Layer
TLMM	Top-Level Mode Multiplexer