Qualcomm
TECHNOLOGIES, INC

MDM9x35 Modem GPIO Software

User Guide

80-NH740-24 A

September 25, 2013

Submit technical questions at:
https://support.cdmatech.com/

Confidential and Proprietary – Qualcomm Technologies, Inc.

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

Contents

Figures

Tables

Revision history

Revision	Date	Description
A	Sep 2013	Initial release

1 Introduction

1.1 Purpose

This document describes AMSS 9635 customizations that can be made to the customer's development platform. This information is useful during initial development of the hardware ASIC and to become familiar with AMSS 9635 software.

1.2 Scope

This document is for software engineers who want to use or configure MDM9x35 GPIOs.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

1.4 References

Reference documents, are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1
Q3	*MDM9x35 Software Interface for OEMs*	80-NH377-2X

1.5 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 Service website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, refer to [Q1].

2 Hardware and Software Overview

2.1 Hardware overview

There are 90 GPIOs in the MDM9x35 ASIC. This section provides a brief hardware overview of the GPIOs.

2.1.1 GPIO pad structure

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

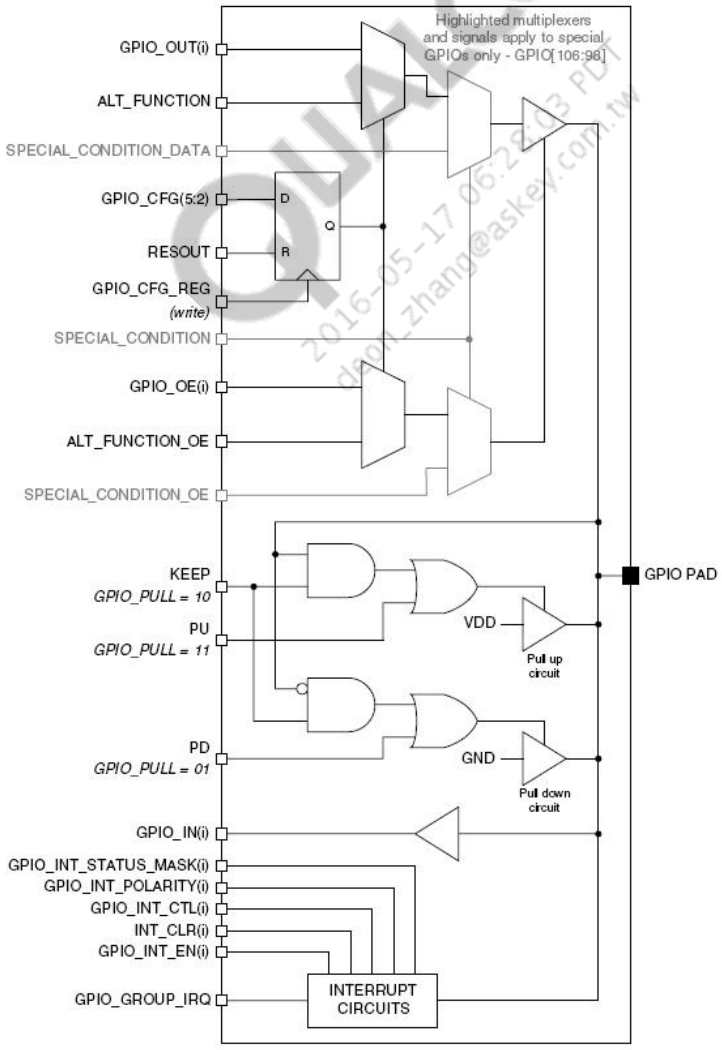

Figure 2-1 Conceptual structure diagram of GPIO pad

As shown, a GPIO pin can be configured as a general input, output, or one of several alternate functions. When used as 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 MDM out of deep sleep (XO shutdown or VDD minimization). Also, a GPIO can be configured to have internal pull (up, down, or no pull) circuits and different driving strength.

2.1.2 GPIO registers

This section describes some important GPIO registers. For complete information on GPIO registers, see the Top-Level Mode Multiplexer (TLMM) and GPIO section of [Q3].

2.1.2.1 Configuration

The configuration of a GPIO is controlled by TLMM_GPIO_CFGn, where n = 0...89 registers. Each register controls the configuration of one GPIO. Table 2-1 shows the GPIO_CFGn register bit field.

Table 2-1 TLMM_GPIO_CFGn register bit field

Bits	Name	Description
31:10	Reserved	Reserved field
10	GPIO_HIHYS_EN	Controls the hihys_en for GPIO n
9	GPIO_OE	Control 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. Values are: ▪ 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 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 (0 = GPIO mode. See [Q3] for a complete list of alternate functions.
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. Values are: ▪ 00 – No pull ▪ 01 – Pull Down (PD) ▪ 10 – Keeper ▪ 11 – Pull Up (PU)

2.1.2.2 Input and output

When the FUNC_SEL of a GPIO is set to 0, this means that 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 TLMM_GPIO_CFGn register. When this bit is set to 1, it is an output pin. When this bit is set to 0, it is an input pin.

When set as output, a GPIO's state is controlled by TLMM_GPIO_IN_OUTn, where n = 0...89 registers. When set as input, a GPIO's state can be read from the same register. Table 2-2 shows the TLMM_GPIO_IN_OUTn register bit field.

Table 2-2 TLMM_GPIO_IN_OUTn 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 you to read the state of an input pin

2.1.2.3 Interrupt

All GPIOs can be set up as interrupt sources. GPIO interrupt is controlled by registers TLMM_GPIO_INTR_CFGn, where n = 0...89. Table 2-3 shows the TLMM_GPIO_INTR_CFGn register bit field.

Table 2-3 TLMM_GPIO_INTR_CFGn bit field

Bit	Name	Description
31:4	Reserved	Reserved field
8	DIR_CONN_EN	Tells the TLMM that GPIO[n] is being used as a direct connect interrupt. Values are: ▪ 0 – DISABLE (Disable the GPIO as direct connect interrupt) ▪ 1 – ENABLE (Enable the GPIO as direct connect interrupt)
7:5	Reserved	Reserved field
4	INTR_RAW_STATUS_EN	Enables the RAW status for the summary. This is a power-saving mechanism. Leave this disabled unless it is needed. Values are: ▪ 1 – Enable ▪ 0 – Disable
3:2	INTR_DECT_CTL	Controls the edge or level detection of the interrupt controller. Values are: ▪ 0x0 – LEVEL (level-sensitive) ▪ 0x1 – POS_EDGE (positive edge sensitive) ▪ 0x2 – NEG_EDGE (negative edge sensitive) ▪ 0x3 – DUAL_EDGE (sensitive to both edges)
1	INTR_POL_CTL	Controls the polarity detection of the interrupt controller. Polarity 1 corresponds to active high and Polarity 0 corresponds to active low. Values are: ▪ 0 – POLARITY_0 ▪ 1 – POLARITY_1

Bit	Name	Description
0	INTR_ENABLE	Controls if this GPIO will generate a summary interrupt. Values are: ▪ 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 summary interrupt status for GPIO[n] is recorded in TLMM_GPIO_INTR_STATUSn, where n = 0...89 registers. When read, this register returns the status of the corresponding GPIO's interrupt status. The interrupt is active when it returns 1. The interrupt is not active when it returns 0. To clear the GPIO interrupt status, write a 0 into this register. 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 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, and achieves the goal of better portability and easy maintenance.

2.2.1 Code locations

The following lists the locations of some driver code that is referred to in this document:

- TLMM code – \core\systemdrivers\tlmm\
- GPIO interrupt driver code – \core\systemdrivers\GPIOInt\
- Header files – \core\api\systemdrivers\

3 GPIO HAL Layer

3.1 Introduction to HAL

Software development has typically followed and lagged behind hardware development. The overall process takes a long time because the 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 MDM9x35 was redesigned 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 the hardware is finalized.

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

With the addition of HAL and DAL, the overall software architecture is described in the block diagram of Figure 3-1.

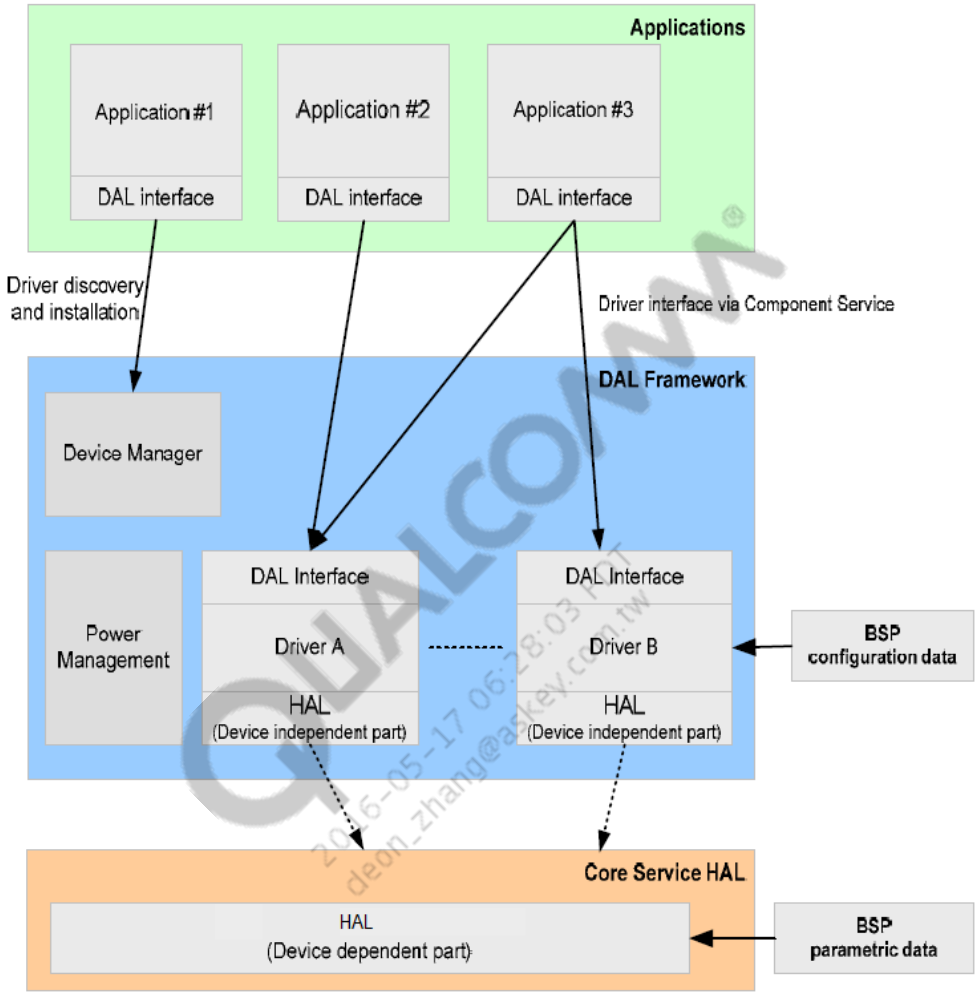

Figure 3-1 Detailed view of software architecture with HAL and DAL

In this new software architecture, the application software invokes the services of device drivers through DAL, and device drivers access the hardware via 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.

The following lists a few characteristics of the HAL software architecture:

- The HAL APIs, i.e. the interface part, are immutable, although 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 following common functions:
  - Initialization
  - Reset
  - Save
  - Restore

3.2 TLMM interface of HAL

This section describes the HAL interface for TLMM.

- The interface part of TLMM HAL is described in HALtlmm.h.
- The actual implementation code of TLMM HAL is under \core\systemdrivers\tlmm\hw\.

3.2.1 Type definitions

HAL TLMM has some basic type definitions related to GPIO properties:

```
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;
```

```
typedef enum
{
     HAL_TLMM_LOW_VALUE  = 0,
     HAL_TLMM_HIGH_VALUE = 1
}HAL_tlmm_ValueType;

/*
 * 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 different direction, pull, and driving strength types of a GPIO's configuration.

3.2.2 Macros

There are a few HAL macros that are visible to the TLMM driver. These macros are used to extract the properties of GPIO configuration:

```
#define HAL_GPIO_NUMBER(config) (((config)&0x3FF0)>>4)
#define HAL_GPIO_OUTVAL(config) (((config)&0x1E00000)>>0x15)
#define HAL_DRVSTR_VAL(config) (((config)&0x1E0000)>>17)
#define HAL_PULL_VAL(config) (((config)&0x18000)>>15)
#define HAL_DIR_VAL(config) (((config)&0x4000)>>14)
#define HAL_FUNC_VAL(config)  ((config)&0xF)
```

- HAL_GPIO_NUMBER – Returns the GPIO number based on configuration data
- HAL_GPIO_OUTVAL – Returns the OUTPUT value
- HAL_DRVSTR_VAL – Returns driving strength
- HAL_PULL_VAL – Returns PULL direction
- HAL_DIR_VAL – Returns direction information, INPUT or OUTPUT
- HAL_FUNC_VAL – Returns the function value; zero is general purpose, nonzero is alternate function

3.2.3 Functions

This section describes some HAL TLMM APIs. These functions are listed here to help you understand the GPIO driver described in Chapter 4, and the use of these functions should be limited to the GPIO driver only, i.e., 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 is used to configure the specified GPIO to the requested configuration:

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

A few 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 the function to configure keysense GPIOs, etc. These functions are listed as follows. The following API can be used to configure a group of GPIOs.

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

The following API can be used to obtain the current GPIO configuration from GPIO_CFGn register:

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

3.2.3.3 Read and write

The following function is used to read values from GPIOs' GPIO_IN registers. One GPIO is read each time this function is called, and the value is returned as a Boolean type: TRUE means HIGH state and FALSE means 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. As with the HAL_tlmm_ReadGpio, the value to be written is stored in a Boolean-type parameter.

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

In the TLMM driver, we divide GPIOs into different groups e.g., GPIO_GROUP_AUDIO_PCM, GPIO_GROUP_USB, GPIO_GROUP_SDC1, etc. Sometimes, we need 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 TLMM is used to set certain TLMM register so that the associated GPIOs are set to a particular special function.

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

3.3 GPIO interrupt interface of HAL

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

3.3.1 Basic definitions

This definition enumerates the four 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 noteworthy HAL GPIO interrupt APIs. As with HAL TLMM APIs, these functions are listed here to help you understand the GPIO interrupt driver code. Use of these functions should be limited to the TLMM driver only, i.e., application software should not call these functions directly.

3.3.2.1 Initialization

This function initializes the HAL's GPIO interrupt controller:

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

3.3.2.2 Restore and save

This function saves the current hardware context:

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

This function restores the hardware context saved by 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 is used to set 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

There are some HAL functions that respond to certain driver code inquiries. 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 more GPIO interrupts are pending.

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

This function returns the number of supported GPIO interrupts on this platform, which may be less than HAL_GPIOINT_NUM:

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

This function is used to get 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 whether the given interrupt is enabled or not:

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

4 GPIO DAL Driver

4.1 Introduction

This chapter describes the API for QTI's DAL TLMM (DALTLMM).The DAL provides an interface to the TLMM driver. All drivers can use the TLMM Device Driver Interface (DDI) 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 clients using the TLMM driver.

The goal of these APIs is to provide a uniform and efficient interface to the driver that is independent of the OS and the processor on which the driver resides. This paves the way for quick integration of QTI ASICs into customers' products.

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-independent OS-independent.

4.2 GPIO configurations

A GPIO generally has two types of configurations; runtime and sleep.

4.2.1 Runtime configuration

The MDM9x35 does not rely on the TLMM driver to initialize GPIO runtime. It is the responsibility of the software module that owns the GPIO to configure it into the appropriate state. It also does not save the current GPIO configuration data in software; since each GPIO has its own configuration register, GPIO configuration information is obtained by reading the GPIO register directly.

4.2.2 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 MDM sleep. This section contains some general guidelines on deciding the sleep configuration.

When the MDM goes into sleep, avoid the following two scenarios to avoid leakage current on a GPIO pad:

- Conflict between MDM and external device – GPIO should be driven by either MDM or external device, and only one of them.
- Floating pin – GPIO should have a definite state (either HIGH or LOW) during sleep. If a GPIO is left floating around Vdd/2, the quiescent current in the IC circuit is the highest.

To avoid these two scenarios, we need to examine how GPIOs are used and how a peripheral device behaves during sleep, e.g., 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 logic state and internal pull. The internal pull is weak (~100 kΩ), and it determines the GPIO state only when neither the MDM nor the external device drives the pin. A GPIO should not have a HIGH state during sleep when PULL_HIGH is defined for the pin.

The sleep configuration of GPIOs is saved in boot build's TLMMChipset.xml file.

4.2.3 GPIO configuration timeline

This section describes how a GPIO can go through different configurations during MDM operation.

1. Once the MDM boots up, GPIOs are initialized in their default hardware state.
2. The TLMM driver initializes GPIOs with their sleep configuration if DALTLMM_PRG_YES is specified for the GPIO in TLMMChipset.xml file. This is to ensure that any unused GPIOs will remain in their low power state once the MDM comes up.
   - If it is desirable to avoid this type of initial configuration on a GPIO, DALTLMM_PRG_NO can be specified for that GPIO in TLMMChipset.xml file.
3. 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.
4. When the software module that owns the GPIO goes into sleep, it needs to apply the sleep configuration on the GPIO to conserved power. This can be done by calling the TLMM API, e.g., DalTlmm_ConfigGpio, with parameter DAL_TLMM_GPIO_DISABLE. The TLMM will automatically apply the sleep configuration stored in TLMMChipset.xml of the boot build.

4.3 DALTLMM data definitions

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

4.3.1 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)
```

Descriptions of the parameters are as follows:

- gpio – GPIO number associated with this signal
- func – GPIO function associated with this signal
  - 0 – General purpose
  - Nonzero – Alternate functions
- dir – Direction (input, output) when the GPIO is set as a general purpose pin
- pull – Pull type (pull up, pull down, no pull, etc.)
- drvstr – Drive strength for this signal

The following example generates the configuration data for 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 you to specify an additional outval that defines the OUTPUT state (HIGH or LOW) 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 needed when defining the low power configuration of an output GPIO.

4.3.2 GPIO state variable

This variable is the main state structure that contains a lot of information on GPIOs of a particular target:

```
DALTLMMStateType* pgtState;
```

It contains the sleep configuration data that is loaded from the TLMMChipset.xml file and whether a GPIO is currently active. The TLMM driver does not save GPIO active configuration any more since each GPIO has its own configuration register in MDM9x35.

4.4 DALTLMM interface description

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

4.4.1 DalTlmm_ConfigGpio

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

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

NOTE: Create a TLMM device handle before calling any DALTLMM APIs.

The following example shows the DALTLMM API configuring GPIO0 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);
```

4.4.2 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 of 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
);
```

4.4.3 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
);
```

4.4.4 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
);
```

4.4.5 DalTlmm_GetGpioStatus

This API returns the current status (active or inactive, i.e., whether the GPIO is in sleep configuration) of a GPIO:

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

4.4.6 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.

4.4.7 DalTlmm_SetPort

This API can be used to set a specific TLMM port, e.g., UIM, for certain GPIOs:

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

4.4.8 DalTlmm_GpioIn

This function returns the value corresponding to the input value of the GPIO, i.e., DAL_GPIO_HIGH_VALUE or DAL_GPIO_LOW_VALUE:

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

4.4.9 DalTlmm_GpioOut

This function drives an output GPIO pin to HIGH or LOW state:

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

Acceptable values are DAL_GPIO_HIGH_VALUE and DAL_GPIO_LOW_VALUE. Note that values will not take effect if the ownership is not configured correctly.

The following API can be used to set 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

5.1 Overview

Each GPIO can act as an interrupt source. The GPIO interrupt driver provides facility for configuring GPIO interrupt's trigger type and enabling/disabling GPIO interrupt.

5.2 Data defintions

5.2.1 Trigger and ISR types

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

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

The following example 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.

5.2.2 DAL GPIO interrupt data structure

The following data structure contains valuable debug information about 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;
```

Most useful information for debugging is in the state and the log.

5.3 DAL GPIO interrupt APIs

This section describes some DAL GPIO interrupt APIs.

5.3.1 GPIO interrupt register and deregister

This API allows 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)
```

For the purpose of deregistering and disabling a GPIO interrupt, the following API is provided:

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

5.3.2 GPO interrupt trigger

The following API can be used to set the desired trigger type for a GPIO interrupt without providing an ISR:

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

5.3.3 Is interrupt enabled

This state returns whether the GPIO interrupt has been enabled:

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

5.3.4 Is interrupt pending

This state returns whether the GPIO interrupt is pending, i.e., enabled and has been triggered:

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

5.3.5 Example code

The following example registers an ISR and changes 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);
```

5.4 Wake-up interrupts

As described in Section 2.1.1, only a subset of GPIOs are capable of waking up the MDM from deep sleep (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 will recognize this and forward it to the RPM so that this interrupt can be enabled in the MPM when the RPM executes deep sleep.

This process of identifying and registering wake-up interrupts is transparent to user code in the modem subsystem, and is done automatically by the GPIO interrupt driver. There is no additional configuration necessary in the modem software code, although you should be aware of which GPIOs can act as wake-up sources. This information can be found in [Q3].