www.neoway.com

Neoway 有方

MSM8909 GPIO/I2C/UART/SPI Configuration Guide

Version 1.0

有 无 线 方 精 彩

LET'S ENJOY WIRELESS LIFE

Notice

This document provides guide for users to use the N1 series.

This document is intended for system engineers (SEs), development engineers, and test engineers.

The information in this document is subject to change without notice due to product version update or other reasons.

Every effort has been made in preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Neoway provides customers complete technical support. If you have any question, please contact your account manager or email to the following email addresses:

Sales@neoway.com

Support@neoway.com

Website: http://www.neoway.com

Revision Record

Issue	Changes	Date
V1.0	Initial draft	2016-06

Contents

1 GPIO

1.1 Hardware Overview

MSM8909 provides 112 GPIOs.

1.1.1 GPIO Registers

This section describes the GPIO configuration registers of the MSM8909 chipset.

Table 1-1 GPIO address

Register	Address	Reset State
TLMM_GPIO_CFGn, n =[0..111]	0x01000000 + 0x00000000 + 0x1000 * (n)	0x0000001
TLMM_GPIO_IN_OUTn , n=[0..111]	0x01000000 + 0x00000004 + 0x1000 * (n)	0x0000000

Table 1-2 GPIO CFG list

Bit	Name	Description
10	GPIO_HIHYS_EN	Control the HIHYS_EN for GPIO[n]
9	GPIO_OE	Control the Output Enable (OE) for GPIO[n] when in GPIO mode
8:6	DRV_STRENGTH	Controls the GPIO pad drive strength. This applies regardless of the FUNC_SEL field selection. 0: DRV_2_MA 1: DRV_4_MA 2: DRV_6_MA 3: DRV_8_MA 4: DRV_10_MA 5: DRV_12_MA 6: DRV_14_MA 7: DRV_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.
1:0	GPIO_PULL	The pad can be configured to employ an internal weak pull up, pull down, or keeper function. This applies regardless of the FUNC_SEL field selection. 0x0: NO_PULL (Disable all pull) 0x1: PULL_DOWN (Weak Pull-down) 0x2: KEEPER (Weak keeper) 0x3: PULL_UP (Weak Pull-up)

1.1.2 Interrupt Configuration

Table 1-3 GPIO interrupt CFG

Register	Address	Reset State
TLMM_GPIO_INTR_CFGn , n=[0..121]	0x01000000 + 0x00000008 + 0x1000 * (n)	0x000000E2
TLMM_GPIO_INTR_STATUSn , n=[0..121]	0x01000000 + 0x0000000C + 0x1000 * (n)	0x00000000

Table 1-4 INT CFG

Bit	Name	Description
8	DIR_CONN_EN	0: DISABLE 1: ENABLE
7:05	TARGET_PROC	0: WCSS 1: SENSORS 2: LPA_DSP 3: RPM 4: KPSS 5: MSS 6: TZ 7: NONE
4	INTR_RAW_STATUS_EN	0: DISABLE 1: ENABLE
3:02	INTR_DECT_CTL	0: LEVEL 1: POS_EDGE (Positive-edge sensitive) 2: NEG_EDGE (Negative-edge sensitive) 3: DUAL_EDGE (Sensitive to both edges)
1	INTR_POL_CTL	0: POLARITY_0 1: POLARITY_1
0	INTR_ENABLE	0: DISABLE 1: ENABLE

Table 1-5 GPIO wakeup interrupt

Register	Address	Reset State
TLMM_MPM_WAKEUP_INT_EN_0	0x01000000 + 0x00100008	0x00000000
TLMM_MPM_WAKEUP_INT_EN_1	0x01000000 + 0x0010000C	0x00000000

1.2 Configuring GPIO

1.2.1 Configuring GPIO in Bootloader

Call **gpio_tlmm_config()**.

```
bootable/bootloader/lk/platform/msm8909/gpio.c
void gpio_tlmm_config(uint32_t gpio, uint8_t func,
                      uint8_t dir, uint8_t pull,
                      uint8_t drvstr, uint32_t enable)
{
  uint32_t val = 0;
  val |= pull;
  val |= func << 2;
  val |= drvstr << 6;
  val |= dir << 9;
  writel(val, (uint32_t *)GPIO_CONFIG_ADDR(gpio));
  return;
}
```

1.2.2 Configuring GPIO in the Linux Kernel

This section describes how to configure MSM8909 GPIO in the Linux kernel.

For more details, refer to **Documentation/devicetree/bindings/pinctrl/msm-pinctrl.txt**.

1. Define a pin controller node to the **MSM8909-pinctrl.dtsi** file.

```
&soc {
   tlmm_pinmux: pinctrl@1000000{
   Client1_pins {
          /* Uses general purpose pins */
          qcom,pins = <&gp 0>, <&gp 1>;
qcom,num-grp-pins = <2>;
          /* function setting as specified in board-<soc>-gpiomux.c */
          qcom,pin-func = <1>;
          label = "client1-bus";
          /* Active configuration of bus pins */
             Client1_default: client1_default {
          /* Property names as specified in  pinctrl-bindings.txt /
          drive-strength = <8>; /* 8 MA */
          bias-disable; /* No PULL */
          };
};
```

2. Modify the client node on the device tree (**msm8909.dtsi** or **msm8909-<board>.dtsi**).


```
    Soc {
Client1 {
/*Please note default state is programmed by the kernel at the time
the time of kernel bootup.No driver changes necessary, since at
probe time the default state would already be programmed in TLMM */
    pinctrl-states = “default”;
    /* Use phandle referrence to default configuration node */
    pinctrl-0 = <&Client1_default>;
};
};
```

1.2.3 Process

Check **drivers/i2c/muxes/i2c-mux-pinctrl.c**.

1. Obtain GPIO information:

```
of_property_read_string_index(np, "pinctrl-names", i, out)
```

2. Parse pin control:

```
devm_pinctrl_get()
```

3. Set states by **pinctrl_lookup_state()** and **pinctrl_select_state()**.

```
for (i = 0; i < mux->pdata->bus_count; i++) {
    mux->states[i] = pinctrl_lookup_state(mux->pinctrl,
           mux->pdata->pinctrl_states[i]);
      …
 }
…
for (i = 0; i < mux->pdata->bus_count; i++) {
  u32 bus = mux->pdata->base_bus_num ?
        (mux->pdata->base_bus_num + i) :0;
    mux->busses[i] = i2c_add_mux_adapter(mux->parent, &pdev->dev,
              mux, bus, i, 0,
              i2c_mux_pinctrl_select,
              deselect);
}
```

4. Add DTS.

```
  i2c0_active {
    /* CLK, DATA */
    qcom,pins = <&gp 7>, <&gp 6>;
    qcom,num-grp-pins = <2>;
    qcom,pin-func = <3>;
    label = "i2c0-active";
      /* active state */
```



```
    i2c_default:default {
      drive-strength = <2>; /* 2 MA */
      bias-disable = <0>; /* No PULL */
    };
  };
```


2 I2C

This chapter describes embedded I2C and its configuration in the core.

2.1 Hardware Overview

2.1.1 Qualcomm Unified Peripheral (QUP)

The Qualcomm Universal Peripheral (QUP) Serial Engine provides a general-purpose data path engine to support multiple mini cores. Each mini core implements protocol-specific logic. The common FIFO provides a consistent system IO buffer and system DMA model across widely varying external interface types. For example, one pair of FIFO buffers can support Serial Peripheral Interface (SPI) and I2C mini-cores independently.

2.1.2 I2C Core

On the MSM8909 chipset, the Linux I2C driver supports Standard mode (100kHz) and Fast mode (up to 400 MHz). The following key features have been added:

- BAM integration
- Support for I2C tag version

To match the labeling in the software interface manual, each QUP is identified by a BLSP core and QUP core (1 to 6).

NOTE

The MSM8909 chipsets contain a single BLSP core.

Table 2-1 QUP physical address

BLSP Hardware ID	QUP Core	Physical Address
BLSP1	BLSP1 QUP 1	0x78B5000
BLSP1	BLSP 1 QUP 2	0x78B6000
BLSP1	BLSP 1 QUP 3	0x78B7000
BLSP1	BLSP 1 QUP 4	0x78B8000
BLSP1	BLSP 1 QUP5	0x78B9000
BLSP1	BLSP 1 QUP 6	0x78BA000

Table 2-2 QUP IRQ Number

BLSP Hardware ID	QUP Core	IRQ Number
BLSP1	BLSP 1 QUP 1	95
BLSP1	BLSP 1 QUP 2	96
BLSP1	BLSP 1 QUP 3	97
BLSP1	BLSP 1 QUP 4	98
BLSP1	BLSP 1 QUP 5	99
BLSP1	BLSP 1 QUP 6	100

2.2 Configuring Kernel I2C

This section describes how to configure a QUP core as an I2C in the kernel.

By default, Qualcomm pre-configure BLSP1_QUP2 and BLSP1_QUP5 as an I2C. For more details, see **/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi** and **Documentation/devicetree/bindings/i2c/i2c-msm-v2.txt**.

2.2.1 Changing Code

The following steps are required to configure and use any of the QUP cores (specifically, BLSP1_QUP2) as an I2C device.

1. Create a device tree node. Modify the following file to add a new device tree node.

/kernel/arch/arm/boot/dts/msm8909.dts

```
        i2c_0: i2c@78b6000 { /* BLSP1 QUP2 */
                compatible = "qcom,i2c-msm-v2";
                reg-names = "qup_phys_addr", "bam_phys_addr";
                reg = <0x78b6000 0x1000>,
<0x7884000 0x23000>;
                interrupt-names = "qup_irq", "bam_irq";
                interrupts = <0 96 0>, <0 238 0>;
                clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
<&clock_gcc clk_gcc_blsp1_qup2_i2c_apps_clk>;
                clock-names = "iface_clk", "core_clk";
                qcom,clk-freq-out = <400000>;
                qcom,clk-freq-in  = <19200000>;
                pinctrl-names = "i2c_active ", "i2c_sleep";
                pinctrl-0 = <&i2c_2_active>;
                pinctrl-1 = <&i2c_2_sleep>;
                qcom,noise-rjct-scl = <0>;
```



```
                qcom,noise-rjct-sda = <0>;
                qcom,bam-pipe-idx-cons = <6>;
                qcom,bam-pipe-idx-prod = <7>;
                qcom,master-id = <86>;
    };
```

2. Modify the following file to add a new clock node.

Project_Root/drivers/clk/qcom/clock-gcc-8909.c

```
    /* Clock lookup */
    static struct clk_lookup msm_clocks_lookup[] = {
  //Add node to BLSP1 AHB Clock
  CLK_LIST(gcc_blsp1_ahb_clk),
  /*
  Add a node to QUP Core clock.
  Note:In clock regime QUP cores are label #1 to #6.
  */
  CLK_LIST(gcc_blsp1_qup1_i2c_apps_clk),
```

3. Modify the following file to set GPIO.

arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi

```
 i2c0_active {
    /* CLK, DATA */
    qcom,pins = <&gp 7>, <&gp 6>;
    qcom,num-grp-pins = <2>;
    qcom,pin-func = <3>;
    label = "i2c0-active";
    /* active state */
    i2c_default:default {
      drive-strength = <2>; /* 2 MA */
      bias-disable = <0>; /* No PULL */
    };
  };
 i2c0_suspend {
    /* CLK, DATA */
    qcom,pins = <&gp 7>, <&gp 6>;
   qcom,num-grp-pins = <2>;
    qcom,pin-func = <0>;
    label = "i2c0-suspend";
    /*suspended state */
    i2c_sleep:sleep {
      drive-strength = <2>; /* 2 MA */
      bias-disable = <0>; /* No PULL */
    };
  };
```


4. Register the GPIO.

```
Kernel/driver/i2c/buses/i2c-msm-v2.c
static int i2c_msm_rsrcs_gpio_pinctrl_init(struct i2c_msm_ctrl *ctrl)
{
    ctrl->rsrcs.pinctrl = devm_pinctrl_get(ctrl->dev);
    if (IS_ERR_OR_NULL(ctrl->rsrcs.pinctrl)) {
      dev_err(ctrl->dev, "failed to get pinctrl\n");
      return PTR_ERR(ctrl->rsrcs.pinctrl);
    }
    ctrl->rsrcs.gpio_state_active
      = pinctrl_lookup_state(ctrl->rsrcs.pinctrl,
         PINCTRL_STATE_DEFAULT);
    if (IS_ERR_OR_NULL(ctrl->rsrcs.gpio_state_active))
      dev_info(ctrl->dev, "can not get default pinstate\n");
    ctrl->rsrcs.gpio_state_suspend
      = pinctrl_lookup_state(ctrl->rsrcs.pinctrl,
         PINCTRL_STATE_SLEEP);
    if (IS_ERR_OR_NULL(ctrl->rsrcs.gpio_state_suspend))
      dev_info(ctrl->dev, "can not get sleep pinstate\n");
    return 0;
}
```

5. Verify the I2C bus.

   Ensure that the bus is registered. If all information is entered correctly, you should see the I2C bus registered under **/dev/i2c-#,** where the cell-index matches the bus number.

```
adb shell --> Get adb shell

cd /dev/
ls i2c* --> to List all the I2C buses
root@android:/dev # ls i2c*
ls i2c*
i2c-0
i2c-10
i2c-2
```

2.2.2 Testing

You can compile and run the program to test the I2C bus in **/vendor/qcom/proprietary/kernel-tests/i2c-test/**.

- If the I2C bus is correctly programmed and the slave device responds, the following output appears:

```
root@android:/data # ./i2c-test
./i2c-test
```



```
Device Open successfull [3]
I2C RDWR Returned 1
```

- If an error occurs, the following output appears:

```
./i2c-test
Device Open successfull [3]
I2C RDWR Returned -1
```

- If I2C RDWR returns -1, check the kernel log for the driver error message. The following error message indicates that the slave device did not send an acknowledgment. The bus is correctly configured and at least the start bit and address bit were sent from the bus, but the slave refused it and did not acknowledge it.

```
[ 6131.397699] qup_i2c f9927000.i2c:I2C slave addr:0x54 not connected
```

  At this point, the debugging should focus on the slave device to make sure it is correctly powered up and ready to accept messages.

- The error message shown below may be due to multiple issues:
  - Invalid software configuration
  - Invalid hardware configuration
  - Slave device issues

```
[ 6190.209880] qup_i2c f9927000.i2c:Transaction timed out, SL-AD = 0x54
[ 6190.216389] qup_i2c f9927000.i2c:I2C Status:132100
[ 6190.221247] qup_i2c f9927000.i2c:QUP Status:0
[ 6190.225857] qup_i2c f9927000.i2c:OP Flags:10
```

2.2.3 Debugging

This section provides debugging tips for situations where the I2C fails for simple read/write operations.

1. Check SDA/SCL idling. Scope the bus to ensure that the SDA/SCL is idling at the high logic level. If it is not idling high, either there is a hardware configuration problem or the GPIO settings are invalid.
2. Set a breakpoint at the line where the error message is coming, for example, at the `Transaction timed out` message.

```
static int
i2c_msm_frmwrk_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[],
    int num)
{
...//Put a breakpoint inside if statement.
  if (!timeout) {
    uint32_t istatus = readl_relaxed(dev->base + QUP_I2C_STATUS);
...
}
```

2.2.4 Registering a Slave Device Using the Device Tree

After the I2C bus is properly verified, you can create a slave device driver and register it with the I2C bus. See the following files for examples:

- For an I2C slave device, refer to **msm8909-qrd.dts**.
- For Synaptic Touch Screen driver registration, refer to **kernel/drivers/input/touchscreen/synaptics_i2c_rmi4.c**.

The following examples show the minimum requirement for properly registering a slave device using the device tree.

1. Create a device tree node. Open the following file and add a device tree node:

   /kernel/arch/arm/boot/dts/qcom/msm8909-qrd.dts

```
&i2c_0 { /* BLSP1 QUP2 */
  synaptics@c {//Slave driver and slave Address
    compatible = "synaptics,rmi4";//Manufacture, model
    reg = <0x0c>; //Slave Address
    interrupt-parent = <&msmgpio>;//GPIO handler
    interrupts = <17 0x2>;//GPIO # will be converted to gpio_irq
    vdd-supply = <&pm8909_l17>;
    vio-supply = <&pm8909_l6>;
    synaptics,reset-gpio = <&msmgpio 16 0x00>;//Pass a GPIO
    synaptics,irq-gpio = <&msmgpio 17 0x00>;
    synaptics,button-map = <139 102 158>;
    synaptics,i2c-pull-up;
    synaptics,reg-en;
  };
};
```

2. Create or modify the slave driver. The following provides an example of the slave driver.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/i2c.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/gpio.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>
#include <linux/regulator/consumer.h>
#include <linux/string.h>
#include <linux/of_gpio.h>
#ifdef CONFIG_OF //Open firmware must be defined for dts useage
static struct of_device_id qcom_i2c_test_table[] = {
```

```
  { .compatible = "qcom,i2c-test",}, //Compatible node must match dts
  { },
};
#else
#define qcom_i2c_test_table NULL
#endif
//I2C slave id supported by driver
static const struct i2c_device_id qcom_id[] = {
  { "qcom_i2c_test", 0 },
  { }
};
static int i2c_test_test_transfer(struct i2c_client *client)
{
  struct i2c_msg xfer; //I2C transfer structure
  u8 buf = 0x55;  //data to transfer
  xfer.addr = client->addr;
  xfer.flags = 0;
  xfer.len = 1;
  xfer.buf = &buf;
  return i2c_transfer(client->adapter, &xfer, 1);
}

static int __devinit i2c_test_probe(struct i2c_client *client,
    const struct i2c_device_id *id)
{
  int irq_gpio = -1;
  int irq;
  int addr;
  //Parse data using dt.
  if(client->dev.of_node){
    irq_gpio = of_get_named_gpio_flags(client->dev.of_node,
"qcom_i2c_test,irq-gpio", 0, NULL);
  }
  irq = client->irq; //GPIO irq #. already converted to gpio_to_irq
  addr = client->addr; //Slave Addr
  dev_err(&client->dev, "gpio [%d] irq [%d] gpio_irq [%d] Slaveaddr [%x]
\n", irq_gpio, irq,
    gpio_to_irq(irq_gpio), addr);
  //You can initiate a I2C transfer anytime using i2c_client *client
structure
  i2c_test_test_transfer(client);
  return 0;
}
```



```
//I2C Driver Info

static struct i2c_driver i2c_test_driver = {
  .driver = {
    .name  = "qcom_i2c_test",
    .owner  = THIS_MODULE,
    .of_match_table = qcom_i2c_test_table,
  },
  .probe    = i2c_test_probe,
  .id_table      = qcom_id,
};
//Easy wrapper to do driver init
module_i2c_driver(i2c_test_driver);
MODULE_DESCRIPTION("I2C TEST");
MODULE_LICENSE("GPL v2");
```

In the kernel log, the following message indicates the device tree was successfully configured:

```
<3>[   2.670731] qcom_i2c_test 2-0052: gpio [61] irq [306] gpio_irq
    [306]
Slaveaddr [52]
```

3 UART

This chapter describes the Universal Asynchronous Receiver/Transmitter (UART) and explains how to configure it in the boot loader and kernel.

3.1 Hardware Overview

3.1.1 BLSP

BAM Low Speed Peripheral (BLSP) is a new design on target chipsets that replaces the legacy GSBI core and are implemented within the peripheral subsystem of the MSM8909 processor.

Each BLSP block includes six QUP and six UART cores. Bus Access Manager/Module (BAM) is used to move data to/from the peripheral buffers. Each BLSP peripheral is statically connected to a pair of BAM pipes. BLSP supports both BAM and non-BAM-based data transfers.

3.1.2 UART Core

Key features added for the chipset include the following:

- BAM support
- Single-character mode

The UART core is used for transmitting and receiving data through a serial interface. It is used for communicating with other UART protocol devices. Configurations of this mode are primarily defined by the UART_DM_MR1 and UART_DM_MR2 registers.

To match the labeling in the software interface manual, each UART is identified by the BLSP core and UART core (1 to 2). The max transfer rate of the UART core is up to 4M bps.

Table 3-1 UART physical address

BLSP Hardware ID	UART Core	Physical Address (UART_DM_BASE_ADDRESS)
BLSP1	BLSP 1 UART 1	0x78AF000
BLSP1	BLSP 1 UART 2	0x78B0000

Table 3-2 UART IRQ

BLSP Hardware ID	UART Core	IRQ Number
BLSP1	BLSP 1 UART 1	107
BLSP1	BLSP 1 UART 2	108

3.2 Configuring UART

3.2.1 Configuring UART in Bootloader

By default, BLSP UART2 is configured in bootloader. The section describes how to configure other UART for debug log.

Changing Code

1. Enable the UART for debugging.

   Set the **WITH_DEBUG_UART** flag to **TRUE** in the **Project_Root/bootable/bootloader/lk/project/msm8909.mk** file.

   ```
   DEFINES += WITH_DEBUG_UART=1
   ```

2. Set the base address in the following file:

   Project_Root/bootable/bootloader/lk/target/msm8909/init.c

   ```
   void target_early_init(void)
   {
   #if WITH_DEBUG_UART
     /*
     First argument represents the ID (can be any since it's not used)
     Second argument, if it is a GSBI base, must be 0
     Third Argument is the physical address for UART CORE defined in
     /bootable/bootloader/lk/platform/msm8909/include/platform/iomap.h
     */
     uart_dm_init(1, 0, BLSP1_UART2_BASE);//it is uart[0..2] instead of
   uart[1..2]
   #endif
   ```

3. Configure the BLSP AHB and UART core clocks in the following file:

   Project_Root/bootable/bootloader/lk/platform/msm8909/acpuclock.c

   Start the clocks:

   ```
   void clock_config_uart_dm(uint8_t id)
   {
     int ret;
     /*
       NOTE:In clock regime clocks are # from 1 to 2 so UART0 would
       be identified as UART1
     */
     //iface_clk is BLSP clk
     ret = clk_get_set_enable("uart2_iface_clk", 0, 1);
   ```

```
  //core_clock is UART clock.
  ret = clk_get_set_enable("uart2_core_clk", 7372800, 1);
}
```

Register the clocks to the clock system.

The BLSP1_AHB clock is enabled by default.

4. Configure the correct GPIO in the following file:

Project_Root/bootable/bootloader/lk/platform/msm8909/gpio.c

```
void gpio_config_uart_dm(uint8_t id)
{
  /*
  Configure the RX/TX GPIO
  Argument 1:GPIO #
  Argument 2:Function (Please see device pinout for more information)
  Argument 3:Input/Ouput (Can be 0/1)
  Argument 4:Should be no PULL
  Argument 5:Drive strength
  Argument 6:Output Enable (Can be 0/1)
  */
gpio_tlmm_config(5, 2, GPIO_INPUT, GPIO_NO_PULL,
  GPIO_8MA, GPIO_DISABLE);
  gpio_tlmm_config(4, 2, GPIO_OUTPUT, GPIO_NO_PULL,
  GPIO_8MA, GPIO_DISABLE);
}
```

Debugging

After the UART is configured properly, the control platform will print the following information:

```
Android boot loader - UART_DM Initialized!!!
```

If the information is not displayed, debug as follows:

- Set a breakpoint in the bootloader.

  At the line **msm_boot_uart_dm_write (uart_dm_base, data, 44)** of the **uart_dm_init** function in the following file:

  /bootable/bootloader/lk/platform/msm_shared/uart_dm.c

  ```
  Check whether GPIO is configured properly.
  ```

- Check the GPIO configuration register (GPIO_CFGn) to ensure that the GPIO settings are valid.

3.2.2 Configuring Kernel Low-Speed UART

The low-speed UART driver is a FIFO-based UART driver designed to support small data transfer at a slow rate, such as for console debugging or IrDA transfer. The high-speed UART driver is a

BAM-based driver that should be used if a large amount of data is transferred or for situations where a high-speed transfer is required.

By default, BLSP1 UART2 with a base address 0x78B0000 is pre-configured as the low-speed UART.

Changing Code

The following table lists the files used to configure BLSP1 UART1 to use the low-speed UART driver.

File type	Description
Device tree source	/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi
Clock table	The clock nodes need to be added to the DTS file. Project_Root/drivers/clk/qcom/clock-gcc-8909.c
Pinctrl settings	The pin control table is located in the following file: arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi

The following procedure describes how to configure BLSP1 UART2 to use the low-speed UART driver

NOTE

Currently, the ID field located to UART has not been displayed. So the first UART to be registered is ttyHSL0, the second one is ttyHSL1, and so on.

1. Create a device tree node. Modify the following file to add a new device tree node.

/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi

```
blsp1_uart2: serial@78b0000 { // 78b0000 is the UART_DM Base Address
    //manufacture, model of serial  driver
        compatible = "qcom,msm-lsuart-v14";
    //Base address UART_DM and size
    /*
         First Field:0 SPI interrupt (Shared Peripheral Interrupt)
        Second Field:Interrupt #
    Third field:Trigger type, keep 0
    information:/kernel/Documentation/devicetree/bindings/arm/gic.tx
    t
    */
        reg = <0x78b0000 0x200>;
        interrupts = <0 108 0>;
        status = " ok "; //Status OK enables it
        clocks = <&clock_gcc clk_gcc_blsp1_uart2_apps_clk>,
    <&clock_gcc clk_gcc_blsp1_ahb_clk>;
        clock-names = "core_clk", "iface_clk";
```



```
        };
```

2. Set the clock. Modify the following file to add the clock node.

drivers/clk/qcom/clock-gcc-8909.c

```
    /* Clock lookup */
    static struct clk_lookup msm_clocks_lookup[] = {
  //Add node to BLSP1 AHB Clock
  CLK_LIST(blsp1_qup1_uart1_apps_clk_src),
  CLK_LIST(blsp1_qup1_uart2_apps_clk_src),
  /*
  Add a node to QUP Core clock.
  Note:In clock regime UART cores are label #1 to #2.
  */
  CLK_LIST(blsp1_uart1_apps_clk_src),
  CLK_LIST(blsp1_uart2_apps_clk_src),
```

3. Set GPIO. Open the **msm8909-pinctrl.dtsi** file in **kernel/arch/arm/boot/dts/qcom/** to update the settings.

```
pmx-uartconsole {
     qcom,pins = <&gp 4>, <&gp 5>;
     qcom,num-grp-pins = <2>;
     qcom,pin-func = <2>;
     label = "uart-console";
     uart_console_sleep:uart-console {
       drive-strength = <16>;
       bias-disable;
     };
   };
```

Debugging Low-Speed UART

This section describes how to debug low-speed UART.

1. Check the UART registration.

   Ensure that the UART is properly registered with the TTY stack.

   Run the following commands:

```
adb shell -> start a new shell
ls /dev/ttyHSL* -> Make sure UART is properly registered
```

   If you do not see your device, check your code modification to ensure that all the information is defined and correct.

2. Check the internal loopback.

a. Run the following commands to enable loopback:

```
adb shell
```

```
mount -t debugfs none /sys/kernel/debug -> mount debug fs
cd /sys/kernel/debug/msm_serial_hsl    -> directory for Low Speed
UART
echo 1 > loopback.#   -> enable loopback.# is device #
cat loopback.#     -> make sure returns 1
```

b. Open another shell to dump the UART Rx data.

```
adb shell
cat /dev/ttyHSL#  ->Dump any data UART Receive
```

c. Transmit some test data through a separate shell.

```
adb shell
echo "This Document Is Very Much Helpful" > /dev/ttyHSL# ->Transfer
   data
```

- If the loopback works:

  Test message loop appears continuously in the command shell until you exit the cat program. This is because of the internal loopback and how the cat program opens the UART.

  It is safe to assume that the UART is properly configured and only the GPIO settings must be confirmed.

- If loopback does not work:

  i Ensure that the UART is still in the Active state. Open the UART from the shell:

  ```
  adb shell
  cat /dev/ttyHSL#  ->Dump any data UART Receive
  ```

  ii Check the clock settings.

- If loopback works, but there is no signal output. Check the GPIO settings.

3.2.3 Configuring Kernel High-Speed UART

UART_DM can be configured as a BAM-based UART. This driver is designed for high-speed, large data transfers, such as Bluetooth communication.

BLSP BAM Address and IRQ

Each BLSP core has one main BAM hardware block and a specific IRQ.

Table 3-3 BLSP BAM physical address

BLSP Hardware ID	UART Core	Physical Address (BLSP_BAM, IRQ)
BLSP1	BLSP 1 UART[1:2]	0x7884000, 238

BAM Pipe Assignment

Two pipes (consumer and producer) are assigned to each BLSP core for data mover operations.

The UartPipeAssignment identifiers are used to each BLSP UART_DM core pipe.

Table 3-4 BAM Pipe

BLSP Hardware ID	UART Core	Pipe (Consumer, Producer)
BLSP1	BLSP 1 UART1	0,1
BLSP1	BLSP 1 UART2	2,3

Changing Code

The following table lists the files used to configure BLSP1_UART2 as a high-speed UART.

File type	Description
Device tree source	`/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi`
Clock table	The clock nodes need to be added to the DTS file. `Project_Root/drivers/clk/qcom/clock-gcc-8909.c`
Pinctrl settings	The pin control table is located in the following file: `arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi`
ARM TrustZone	`/trustzone_images/core/hwengines/bam/8909/bamtgtcfgdata_tz.h`

1. Create a device tree node. Modify the **/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi** file or any other DTS/TDSI file which contains the following information:

```
uart2 – uart@0x78b0000 { //0x78b0000 is the UART_DM Base address for
BLSP1_UART2
    /*
      UART ID, Recommend OEM to use Large ID so does not conflict with
      Qualcomm configured ID.
      cell-index 102 would represent as /dev/ttyHS102
    */
    cell-index = <102>; //UART ID, Recommend OEM to use Large ID # that
    < 255
    compatible = "qcom,msm-hsuart-v14"; //manufacture, model (must be
    same)
    status = "ok"; //"ok" or "okay" to enable
    /*
    First Row UART_DM Base and Size always 0x1000
    Second Row is BAM address, size always 0x19000
```

```
    For BLSP1 UART0:5 Bam Address = 0xF9904000
    */
    reg = <0x78b0000 0x200>;
interrupts = <0 108 0>;
reg-names = "core_mem", "bam_mem"; //Keep the same names
    /*
    First Field:0 SPI interrupt (Shared Peripheral Interrupt)
    Second Field:Interrupt #
    Third field:Trigger type, keep 0
    For more
information:/kernel/Documentation/devicetree/bindings/arm/gic.txt
    First Row UART_DM IRQ #
    Second Row is BAM IRQ
    For BLSP1 UART0:1 IRQ = 238
    */
    interrupts = <0 108 0>, <0 238 0>;
    interrupt-names = "core_irq", "bam_irq"; //Keep same
    qcom,bam-tx-ep-pipe-index = <2>; //BAM Consumer Pipe
    qcom,bam-rx-ep-pipe-index = <3>; //BAM Producer Pipe
  };
```

For more information, see the device tree document:

/kernel/Documentation/devicetree/bindings/tty/serial/msm_serial_hs.txt.

2. Set the clock. Modify the **drivers/clk/qcom/clock-gcc-8909.c** file to add the node to the UART core clock.

This step is based on the MSM8909 Pre-ES version.

```
     /* Clock lookup */
     static struct clk_lookup msm_clocks_lookup[] = {
    //Add node to BLSP1 AHB Clock
    CLK_LIST(blsp1_qup1_uart1_apps_clk_src),
    CLK_LIST(blsp1_qup1_uart2_apps_clk_src),
    /*
    Add a node to QUP Core clock.
    Note:In clock regime UART cores are label #1 to #2.
  */
CLK_LIST(blsp1_uart1_apps_clk_src),
    CLK_LIST(blsp1_uart2_apps_clk_src),
```

3. Set the correct GPIO. Modify the **kernel/arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi** file to update the setting.

NOTE

This step is based on the MSM8909 Pre-ES version.

```
pmx-uartconsole {
      qcom,pins = <&gp 4>, <&gp 5>,<&gp 6>,<&gp 7>;
      qcom,num-grp-pins = <4>;
      qcom,pin-func = <2>;
      label = "uart-console";
      uart_console_sleep:uart-console {
        drive-strength = <16>;
        bias-disable;
      };
};
```

4. Change TrustZone in the **/trustzone_images/core/hwengines/bam/8909/bamtgtcfgdata_tz.h** file to assign BLSP BAM pipes to the application processors.

   Each BLSP is used by any subsystem (e.g. modem and LPASS). Ensure that the relationships between BAM pipes and application processors specified.

```
/*
  bam_tgt_blsp1_secconfig = BLSP1 Core
  Each Bit You set represent Pipe #.
    For example Bit 0 = Pipe # 0
              Bit 5 = Pipe # 5
  Any bit set on TZBSP_VMID_AP owns by APPs
bam_sec_config_type bam_tgt_ce1_secconfig_8909 =
{
    {
     { 0x800000C3 , TZBSP_VMID_TZ, 0x0, TZBSP_VMID_TZ_BIT},  /* krait
    tz */
     { 0x0000003C , TZBSP_VMID_AP, 0x0, TZBSP_VMID_AP_BIT}  /* krait apps
    */
    },
    {   {0x0}  /* SG not supported*/
    },
    TZBSP_VMID_TZ_BIT,
    0x0 /* IRQ will be routed to EE0 */
};
```

Debugging High-Speed UART

This section describes how to debug high-speed UART.

1. Check the registration. Ensure that the UART is properly registered with the TTY stack by running the following commands:

```
adb shell – start a new shell
ls /dev/ttyHS* – Make sure UART is properly registered
```

If the device does not appear, check your code modification to ensure that all information is defined and correct.

2. Check the internal loopback.

a. Run the following commands to enable loopback:

```
adb shell
mount -t debugfs none /sys/kernel/debug  -> mount debug fs
cd /sys/kernel/debug/msm_serial_hs      -> directory for High Speed
  UART
echo 1 > loopback.#                    -> enable loopback. # is device
  #
cat loopback.#                         -> make sure returns 1
```

b. Open another shell to dump the UART Rx data.

```
adb shell
cat /dev/ttyHS#  ->Dump any data UART Receive
```

c. Transmit some test data through a separate shell.

```
adb shell
echo "This Is A Helpful Document" > /dev/ttyHS# ->Transfer data
```

- If the loopback works:

  Test message loop appears continuously in the command shell until you exit the cat program. This is because of the internal loopback and how the cat program opens the UART.

  It is safe to assume that the UART is properly configured and only the GPIO settings must be confirmed.

- If loopback does not work:

  i Ensure that the UART is still in the Active state. Open the UART from the shell:

  ```
  adb shell
  cat /dev/ttyHSL#  ->Dump any data UART Receive
  ```

  ii Check the clock settings.

- If loopback works, but there is no signal output. Check the GPIO settings.

4 Serial Peripheral Interface (SPI)

This chapter describes the SPI and explains how to configure it in the kernel.

4.1 Hardware Overview

The SPI allows full-/half-duplex, synchronous, serial communication between a master and slave. There is no explicit communication framing, error checking, or defined data word length. Hence, the communication is strictly at the raw bit level.

Key feature:

- Supports up to 48 MHz
- Supports 4 to 32 bits per word of transfer
- Supports a maximum of four Chip Selects (CSes) per bus
- Supports BAM

Table 4-1 Base address

BLSP Hardware ID	QUP Core	Physical Address
BLSP1	BLSP 1 QUP 1	0x78B5000
BLSP1	BLSP 1 QUP 2	0x78B6000
BLSP1	BLSP 1 QUP 3	0x78B7000
BLSP1	BLSP 1 QUP 4	0x78B8000
BLSP1	BLSP 1 QUP 5	0x78B9000
BLSP1	BLSP 1 QUP 6	0x78BA000

Each QUP is identified by a BLSP core and a QUP core (1 to 6).

Table 4-2 QUP IRQ

BLSP Hardware ID	QUP Core	IRQ
BLSP1	BLSP 1 QUP 1	95
BLSP1	BLSP 1 QUP 2	96
BLSP1	BLSP 1 QUP 3	97
BLSP1	BLSP 1 QUP 4	98
BLSP1	BLSP 1 QUP 5	99
BLSP1	BLSP 1 QUP 6	100

4.2 Configuring Kernel SPI

4.2.1 Configuring Kernel Low-Speed SPI

This section describes the steps required to configure and use the BLSP1_QUP3 QUP core as an SPI bus. For more details, see **/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi**.

Changing Code

The following table lists the files used to configure a QUP core as an SPI device in the kernel.

File type	Description
Device tree source	`/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi`
Clock table	The clock nodes need to be added to the DTS file. `Project_Root/drivers/clk/qcom/clock-gcc-8909.c`
Pinctrl settings	The pin control table is located in the following file: `arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi`
TrustZone	`/trustzone_images/core/hwengines/bam/8909/bamtgtcfgdata_tz.h`

1. Create a device tree node. In the `kernel/arch/arm/boot/dts/qcom/msm8909.dtsi` file, add a new device tree node.

```
 spi_0: spi@78b7000 { /* BLSP1 QUP3 */
   compatible = "qcom,spi-qup-v2";
              #address-cells = <1>;
              #size-cells = <0>;
              reg-names = "spi_physical", "spi_bam_physical";
              reg = <0x78b7000 0x600>,
<0x7884000 0x23000>;
              interrupt-names = "spi_irq", "spi_bam_irq";
              interrupts = <0 97 0>, <0 238 0>;
              spi-max-frequency = <19200000>;
              pinctrl-names = "default", "sleep", "cs_default";
              pinctrl-0 = <&spi0_default>;
   pinctrl-1 = <&spi0_sleep>;
   pinctrl-2 = <&spi0_cs_sleep>;
   clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
<&clock_gcc clk_gcc_blsp1_qup1_spi_apps_clk>;
   clock-names = "iface_clk", "core_clk";
              qcom,infinite-mode = <0>;
```



```
    qcom,use-bam;
    qcom,use-pinctrl;
                qcom,ver-reg-exists;
                qcom,bam-consumer-pipe-index = <8>;
                qcom,bam-producer-pipe-index = <9>;
                qcom,master-id = <86>;
    status = "ok";
  };
```

For more information, see **/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt**.

2. Set the clock. Modify the following file to add the clock node.

Project_Root/drivers/clk/qcom/clock-gcc-8909.c

```
    /* Clock lookup */
    static struct clk_lookup msm_clocks_lookup[] = {
  //Add node to BLSP1 AHB Clock
CLK_LIST(gcc_blsp1_ahb_clk),
  /*  Add a node to QUP Core clock. */
  CLK_LIST(blsp1_qup3_spi_apps_clk_src),
```

3. Set GPIO. Modify the following file:

Project_Root/arch/arm/boot/dts/qcom/msm8909-pinctrl.dtsi

```
spi0_active {
     /* MOSI, MISO, CLK */
     qcom,pins = <&gp 8>, <&gp 9>, <&gp 11>;
     qcom,num-grp-pins = <3>;
     qcom,pin-func = <1>;
     label = "spi0-active";
     /* active state */
     spi0_default:default {
       drive-strength = <12>; /* 12 MA */
       bias-disable = <0>; /* No PULL */
     };
    };

     spi0_suspend {
      /* MOSI, MISO, CLK */
      qcom,pins = <&gp 8>, <&gp 9>, <&gp 11>;
      qcom,num-grp-pins = <3>;
      qcom,pin-func = <0>;
      label = "spi0-suspend";
      /* suspended state */
      spi0_sleep:sleep {
        drive-strength = <2>; /* 2 MA */
        bias-disable = <0>; /* No PULL */
```



```
        };
    };
    spi0_cs0_active {
        /* CS */
        qcom,pins = <&gp 10>;
        qcom,num-grp-pins = <1>;
        qcom,pin-func = <1>;
        label = "spi0-cs0-active";
        spi0_cs0_active: cs0_active {
          drive-strength = <2>;
          bias-disable = <0>;
        };
      };

    spi0_cs0_suspend {
        /* CS */
        qcom,pins = <&gp 10>;
        qcom,num-grp-pins = <1>;
        qcom,pin-func = <0>;
        label = "spi0-cs0-suspend";
        spi0_cs0_sleep: cs0_sleep {
          drive-strength = <2>;
          bias-disable = <0>;
        };
     };
```

For more details, see **/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt**.

4. Verify the configurations.

   If all the information was correctly entered, the SPI bus will be registered under **/sys/class/spi_master/spi#**, where the cell-index matches the bus number.

```
adb shell --> Get adb shell
cd /sys/class/spi_master to list all the spi master
root@android:/sys/class/spi_master # ls
ls
spi0
spi6
spi7
```

Registering a Slave Device Using the Device Tree

When the SPI bus is registered, create a slave device driver and register it with the SPI master. For examples of SPI slave devices, see the following files:

- **/kernel/arch/arm/boot/dts/qcom/msm8909.dtsi**

- **/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt**
- **/kernel/Documentation/devicetree/bindings/spi/spi-bus.txt**

The following procedure shows the minimum requirements for registering a slave device.

1. Create a device tree node. Modify the following file to add the new device tree node.

   kernel/arch/arm/boot/dts/qcom/msm8909.dtsi

```
  spi_0: spi@78b7000 { /* BLSP1 QUP3 */
   compatible = "qcom,spi-qup-v2";
              #address-cells = <1>;
              #size-cells = <0>;
              reg-names = "spi_physical", "spi_bam_physical";
              reg = <0x78b7000 0x600>,
<0x7884000 0x23000>;
              interrupt-names = "spi_irq", "spi_bam_irq";
              interrupts = <0 97 0>, <0 238 0>;
              spi-max-frequency = <19200000>;
              pinctrl-names = "default", "sleep", "cs_default";
              pinctrl-0 = <&spi0_default>;
   pinctrl-1 = <&spi0_sleep>;
   pinctrl-2 = <&spi0_cs_sleep>;
   clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
<&clock_gcc clk_gcc_blsp1_qup1_spi_apps_clk>;
   clock-names = "iface_clk", "core_clk";
              qcom,infinite-mode = <0>;
qcom,use-bam;
   qcom,use-pinctrl;
              qcom,ver-reg-exists;
              qcom,bam-consumer-pipe-index = <8>;
              qcom,bam-producer-pipe-index = <9>;
              qcom,master-id = <86>;
   status = "disabled";
 };
```

2. Create or modify the slave device driver. The following provides an example of the slave driver.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/spi/spi.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/gpio.h>
#include <linux/debugfs.h>
#include <linux/seq_file.h>
#include <linux/regulator/consumer.h>
```



```
#include <linux/string.h>
#include <linux/of_gpio.h>
#ifdef CONFIG_OF //Open firmware must be defined for dts useage
static struct of_device_id qcom_spi_test_table[] = {
 { .compatible = "qcom,spi-test",}, //Compatible node must match dts
 { },
};
#else
#define qcom_spi_test_table NULL
#endif

#define BUFFER_SIZE 4<<10
struct spi_message spi_msg;
struct spi_transfer spi_xfer;
u8 *tx_buf; //This needs to be DMA friendly buffer
static int spi_test_transfer(struct spi_device *spi)
{
 spi_message_init(&spi_msg);
 spi_xfer.tx_buf = tx_buf;
 spi_xfer.len = BUFFER_SIZE;
 spi_xfer.bits_per_word = 8;
 spi_xfer.speed_hz = spi->max_speed_hz;
 spi_message_add_tail(&spi_xfer, &spi_msg);
return spi_sync(spi, &spi_msg);
}

static int __devinit spi_test_probe(struct spi_device *spi)
{
 int irq_gpio = -1;
 int irq;
 int cs;
 int cpha,cpol,cs_high;
 u32 max_speed;
 dev_err(&spi->dev, "s\n", __func__);
       //allocate memory for transfer
 tx_buf = kmalloc(BUFFER_SIZE, GFP_ATOMIC);
 if(tx_buf == NULL){
   dev_err(&spi->dev, "s: mem alloc failed\n", __func__);
   return -ENOMEM;
 }
 //Parse data using dt.
 if(spi->dev.of_node){
   irq_gpio = of_get_named_gpio_flags(spi->dev.of_node,
```



```
"qcom_spi_test,irq-gpio", 0, NULL);
 }
 irq = spi->irq;
 cs = spi->chip_select;
 cpha = ( spi->mode & SPI_CPHA ) ? 1:0;
 cpol = ( spi->mode & SPI_CPOL ) ? 1:0;
 cs_high = ( spi->mode & SPI_CS_HIGH ) ? 1:0;
 max_speed = spi->max_speed_hz;
 dev_err(&spi->dev, "gpio [d] irq [d] gpio_irq [d] cs [x] CPHA [x] CPOL
[x] CS_HIGH [x]\n",
   irq_gpio, irq, gpio_to_irq(irq_gpio), cs, cpha, cpol, cs_high);
 dev_err(&spi->dev, "Max_speed [d]\n", max_speed );
//Once you have a spi_device structure you can do a transfer anytime
 spi->bits_per_word = 8;
 dev_err(&spi->dev, "SPI sync returned [d]\n",
    spi_test_transfer(spi));
 return 0;
}

//SPI Driver Info
static struct spi_driver spi_test_driver = {
 .driver = {
   .name  = "qcom_spi_test",
   .owner  = THIS_MODULE,
   .of_match_table = qcom_spi_test_table,
 },
 .probe    = spi_test_probe,
};

static int __init spi_test_init(void)
{
 return spi_register_driver(&spi_test_driver);
}

static void __exit spi_test_exit(void)
{
 spi_unregister_driver(&spi_test_driver);
}

module_init(spi_test_init);
module_exit(spi_test_exit);
MODULE_DESCRIPTION("SPI TEST");
MODULE_LICENSE("GPL v2");
```


3. Verify that the device tree was configured. In the kernel log, the following message indicates the device tree was successfully configured.

```
<3>[    2.503571] qcom_spi_test spi6.0: spi_test_probe
<3>[    2.507305] qcom_spi_test spi6.0: gpio [61] irq [306] gpio_irq
    [306]
                   cs [0] CPHA [1] CPOL [1] CS_HIGH [1]
<3>[    2.516825] qcom_spi_test spi6.0: Max_speed [4800000]
<3>[    2.521932] qcom_spi_test spi6.0: SPI sync returned [0]
```

4.2.2 Configuring Kernel High-Speed SPI

The SPI can operate in FIFO-based mode or Data Mover mode (BAM). If large amounts of data are to be transferred, enable BAM to offload the CPU.

BLSP BAM Address and IRQ

Each BLSP core has one main BAM hardware block and a specific IRQ.

Table 4-3 BLSP BAM physical address

BLSP Hardware ID	QUP Core	BLSP_BAM,IRQ
BLSP1	BLSP1_QUP[1:6]	0x7884000, 238

Table 4-4 BAM pipe assignment

BLSP Hardware ID	QUP Core	PIPE (Input, Output)
BLSP1	BLSP1_QUP_1	0,1
BLSP1	BLSP1_QUP_2	2,3
BLSP1	BLSP1_QUP_3	4,5
BLSP1	BLSP1_QUP_4	6,7
BLSP1	BLSP1_QUP_5	8,9
BLSP1	BLSP1_QUP_6	10,11

Two pipes (consumer and producer) are assigned to each BLSP QUP core for data mover operations. MSM8909 contains only one BLSP core.

Changing Code

The following describes how to enable BAM (Data Mover mode) in the SPI.

1. Modify the device tree. The following example shows the additional fields needed in the DTS node to enable SPI BAM mode.

   For more information, see **/kernel/Documentation/devicetree/bindings/spi/spi_qsd.txt**.

```
    spi_0: spi@78b7000 { /* BLSP1 QUP3 */
   compatible = "qcom,spi-qup-v2";
              #address-cells = <1>;
              #size-cells = <0>;
              reg-names = "spi_physical", "spi_bam_physical";
              reg = <0x78b7000 0x600>,
<0x7884000 0x23000>;
              interrupt-names = "spi_irq", "spi_bam_irq";
              interrupts = <0 97 0>, <0 238 0>;
              spi-max-frequency = <19200000>;
              pinctrl-names = "default", "sleep", "cs_default";
              pinctrl-0 = <&spi0_default>;
   pinctrl-1 = <&spi0_sleep>;
   pinctrl-2 = <&spi0_cs_sleep>;
   clocks = <&clock_gcc clk_gcc_blsp1_ahb_clk>,
<&clock_gcc clk_gcc_blsp1_qup1_spi_apps_clk>;
   clock-names = "iface_clk", "core_clk";
              qcom,infinite-mode = <0>;
  qcom,use-bam;
   qcom,use-pinctrl;
              qcom,ver-reg-exists;
              qcom,bam-consumer-pipe-index = <8>;
              qcom,bam-producer-pipe-index = <9>;
              qcom,master-id = <86>;
   status = "ok";
 };
```

2. Change TrustZone in the **/trustzone_images/core/hwengines/bam/8909/bamtgtcfgdata_tz.h** file to assign BLSP BAM pipe to the application processor.

   Each BLSP is used by any subsystem (e.g. modem and LPASS). Ensure that the relationships between BAM pipes and application processors specified.

```
/*
  bam_tgt_blsp1_secconfig = BLSP1 Core
  Each Bit You set represent Pipe #.
    For example Bit 0 = Pipe # 0
              Bit 5 = Pipe # 5
  Any bit set on TZBSP_VMID_AP owns by APPs
*/
bam_sec_config_type bam_tgt_ce1_secconfig_8909 =
```

```
{
    {
      { 0x800000C3 , TZBSP_VMID_TZ, 0x0, TZBSP_VMID_TZ_BIT},  /* krait
    tz */
      { 0x0000003C , TZBSP_VMID_AP, 0x0, TZBSP_VMID_AP_BIT}  /* krait
    apps */
    },
    {   {0x0}  /* SG not supported*/
    },
    TZBSP_VMID_TZ_BIT,
    0x0 /* IRQ is routed to EE0 */
};
```

