USER'S GUIDE

Apollo4 Family Software Development Kit (SDK)

Ultra-Low Power Apollo SoC Family

A-SOCAP4-UGGA06EN v1.0

Legal Information and Disclaimers

Revision History

Revision	Date	Description
1.0	October 6, 2022	Initial release

Reference Documents

Document ID	Description
A-SOCAP4-GGGA01EN	Apollo4 Family SDK Getting Started Guide
N/A	Apollo4 SDK API Documentation

Table of Contents

List of Tables

List of Figures

SECTION 1

Introduction

This document provides high level guidance on the usage of the various Hardware Abstraction Layer (HAL) functions provided in the Ambiqsuite SDK. This document provides a “bridge” from the Apollo4 Family SDK Getting Started Guide to the Apollo4 SDK API Documentation (Doxygen auto-generated) and the example code.

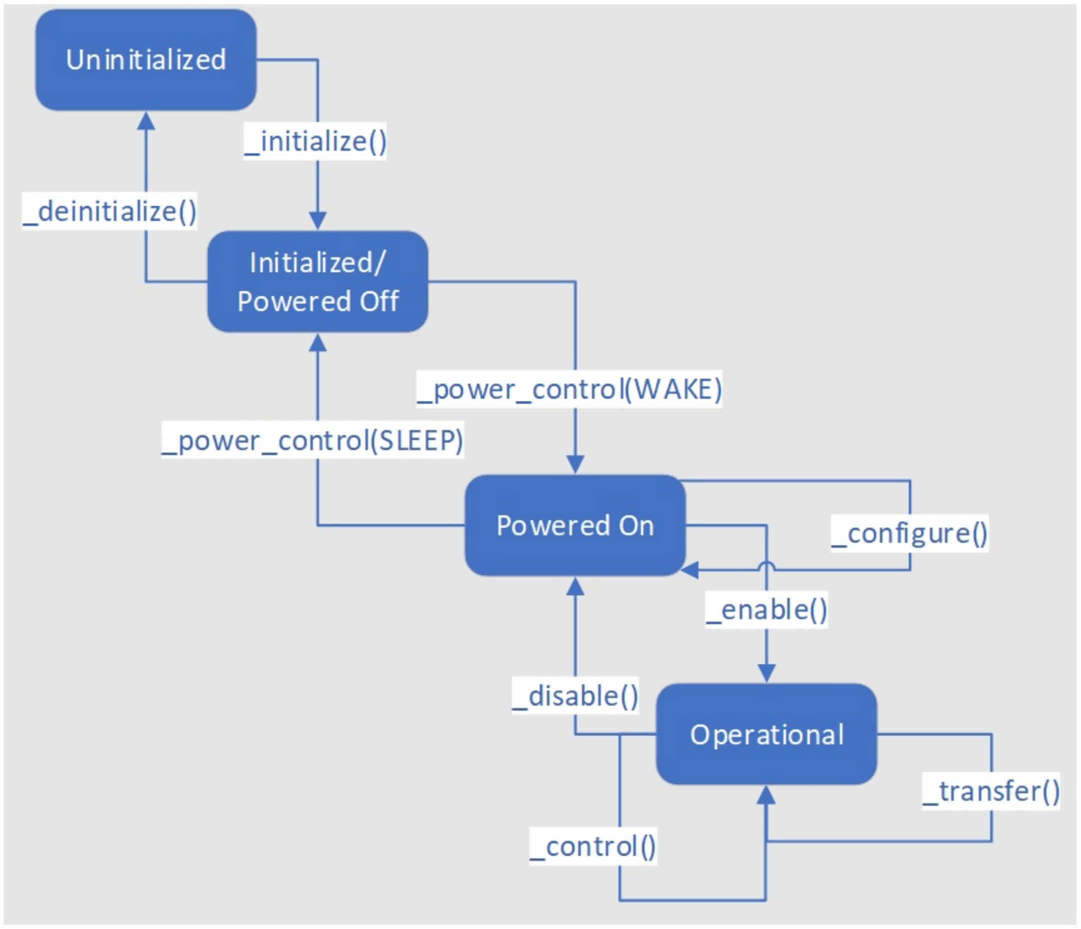
1.1 HAL Driver Uniform API Design

In general, the sequence of initialization of a given HAL driver API follows the state transitions shown in Figure 1-1 on page 8. Many of the hardware blocks in our SoC have multiple instances. Each block starts in the Uninitialized state after system power-up. The **_initialize()** function returns a “handle” to a given instance of a hardware block. The handle is then used from then on for all other function calls related to the instance.

Once initialized, the instance of the block starts without power enabled. Further configuration of the instance requires the **_power_control()** function to be called to power on the instance. Failure to do this will result in faults being detected if other functions try to configure registers. Next, the device can be configured using the **_configure()** function, which puts the block into an operational state[1].

[1] There are often multiple **_configure()** functions for individual blocks depending upon how the HAL is designed. Those details will be covered in subsequent sections of this document.

Figure 1-1: State Transitions of HAL Driver API



Once the instance is operational, the **_control()** function can be used to modify the configuration, if required, during normal operation. Some hardware blocks are "transactional" and will support **_transfer()** functions to allow the user to schedule transactions.

Not shown are a series of functions that are designed to interact with interrupts in the block. This includes:

- **_interrupt_enable** – enables a given set of interrupts
- **_interrupt_disable** – disables a given set of interrupts
- **_interrupt_status** – return the current block interrupts asserted
- **_interrupt_clear** – clear a given set of interrupts
- **_interrupt_service** – provide block specific routine to process common interrupts

SECTION 2

Critical Initialization

2.1 DAXI

The default DAXI configuration is sufficient for most applications as follows:

Apollo4

```
const am_hal_daxi_config_t am_hal_daxi_defaults =
{
    .agingCounter             = 2,
    .eNumBuf                  = AM_HAL_DAXI_CONFIG_NUMBUF_1,
    .eNumFreeBuf              = AM_HAL_DAXI_CONFIG_NUMFREEBUF_2,
};
```

Apollo4 Plus

```
const am_hal_daxi_config_t am_hal_daxi_defaults =
{
    .bDaxiPassThrough         = false,
    .bAgingSEnabled           = true,
    .eAgingCounter            = AM_HAL_DAXI_CONFIG_AGING_4,
    .eNumBuf                  = AM_HAL_DAXI_CONFIG_NUMBUF_32,
    .eNumFreeBuf              = AM_HAL_DAXI_CONFIG_NUMFREEBUF_3,
};
```

2.2 Cache

The default Cache configuration is also sufficient for most applications as follows:

```
const am_hal_cachectrl_config_t am_hal_cachectrl_defaults =
{
    .bLRU                       = 0,
    .eDescript                  = AM_HAL_CACHECTRL_DESCR_1WAY_128B_4096E,
    .eMode                      = AM_HAL_CACHECTRL_CONFIG_MODE_INSTR_DATA,
};
```

In addition, some of the power examples use the following configuration:

```
const am_hal_cachectrl_config_t am_hal_cachectrl_benchmark =
{
    .bLRU                       = 0,
    .eDescript                  = AM_HAL_CACHECTRL_DESCR_1WAY_128B_512E,
    .eMode                      = AM_HAL_CACHECTRL_CONFIG_MODE_INSTR,
};
```

2.3 Low-Power Initialization

The SDK supplies an initialization function call **am_hal_pwrctrl_low_power_init** that provides a uniform setting of default configuration for initial low-power operation. This function does the following:

- Sets the default memory configurations
- Enables the proper clock gating configuration
- Set the default DAXI configuration
- Initialize the SRAM and Extended RAM trims

SECTION 3

Block Specific Initial Operations

This section describes the basics of initialization of the individual hardware blocks within the Apollo4 family. The HAL API calls are shown with a small amount of description for each step/call. This is not a replacement for the complete set of examples that are provided with the AmbiqSuite SDK in the /boards directory. For example, we do not cover how that various functional options for the GPIOs/pins must be configured as those are covered in the examples and BSPs. In addition, we do not show the complete initialization code for interrupt handling.

The following sections appear in alphabetical order based on the block name in the register definitions.

3.1 General Purpose ADC (ADC)

The general-purpose ADC has only one instance, so initialization is as follows:

```
//
// Initialize the ADC and get the handle.
//
if ( am_hal_adc_initialize(0, &g_ADCHandle) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - reservation of the ADC instance
    failed.\n");
}
```

Notice that global variable **g_ADCHandle** is used to store a pointer to the handle that is used for all subsequent function access to the ADC instance.

The next step is to power on the ADC domain as follows:

```
//
// Power on the ADC.
//
if (AM_HAL_STATUS_SUCCESS != am_hal_adc_power_control(g_ADCHandle,
                                                      AM_HAL_SYSCTRL_WAKE,
                                                      false) )
{
    am_util_stdio_printf("Error - ADC power on failed.\n");
}
```

The ADC is configured in three steps. First the ADC block is set up, then at least one of the eight slots is configured for a particular measurement.

```
//
// Set up the ADC configuration parameters. These settings are reasonable
// for accurate measurements at a low sample rate.
//
ADCConfig.eClock             = AM_HAL_ADC_CLKSEL_HFRC;
ADCConfig.ePolarity          = AM_HAL_ADC_TRIGPOL_RISING;
ADCConfig.eTrigger           = AM_HAL_ADC_TRIGSEL_SOFTWARE;
ADCConfig.eClockMode         = AM_HAL_ADC_CLKMODE_LOW_LATENCY;
ADCConfig.ePowerMode         = AM_HAL_ADC_LPMODE0;
ADCConfig.eRepeat            = AM_HAL_ADC_REPEATING_SCAN ;
ADCConfig.eRepeatTrigger     = AM_HAL_ADC_RPTTRIGSEL_INT;
if ( am_hal_adc_configure(g_ADCHandle, &ADCConfig) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - configuring ADC failed.\n");
}

//
// Set up an ADC slot
//
ADCSlotConfig.eMeasToAvg      = AM_HAL_ADC_SLOT_AVG_128;
ADCSlotConfig.ePrecisionMode  = AM_HAL_ADC_SLOT_12BIT;
ADCSlotConfig.eChannel        = AM_HAL_ADC_SLOT_CHSEL_SE1;
ADCSlotConfig.bWindowCompare  = false;
ADCSlotConfig.bEnabled        = true;
ADCSlotConfig.ui32TrkCyc      = AM_HAL_ADC_MIN_TRKCYC;
if (AM_HAL_STATUS_SUCCESS != am_hal_adc_configure_slot(g_ADCHandle, 0,
    &ADCSlotConfig))
{
    am_util_stdio_printf("Error - configuring ADC Slot 0 failed.\n");
}

//
// Set up internal repeat trigger timer
//
am_hal_adc_irtt_config_t      ADCIrttConfig =
{
    .bIrttEnable        = true,
    .eClkDiv            = AM_HAL_ADC_RPTT_CLK_DIV16,
    .ui32IrttCountMax   = 30,
};
am_hal_adc_configure_irtt(g_ADCHandle, &ADCIrttConfig);

//
// Enable internal repeat trigger timer
//
am_hal_adc_irtt_enable(g_ADCHandle);
```

There is another configuration function to set up DMA, which may be used if desired as follows:

```
//
// Configure the ADC to use DMA for the sample transfer.
//
ADCDMAConfig.bDynamicPriority = true;
ADCDMAConfig.ePriority = AM_HAL_ADC_PRIOR_SERVICE_IMMED;
ADCDMAConfig.bDMAEnable = true;
ADCDMAConfig.ui32SampleCount = ADC_SAMPLE_BUF_SIZE;
ADCDMAConfig.ui32TargetAddress = (uint32_t)g_ui32ADCSampleBuffer;
if (AM_HAL_STATUS_SUCCESS != am_hal_adc_configure_dma(g_ADCHandle,
    &ADCDMAConfig))
{
    am_util_stdio_printf("Error - configuring ADC DMA failed.\n");
}
```

In addition to the configuration of the block, the user may also want to enable various interrupts for error conditions, individual samples, or DMA interrupts.

The final step for initialization of the ADC block is to enable the instance as follows:

```
//
// Enable the ADC.
//
if ( am_hal_adc_enable(g_ADCHandle) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - enabling ADC failed.\n");
}
```

3.2 Audio ADC (AUDADC)

The Audio ADC has only one instance but requires a sequence of power-on and configuration steps as outlined in the following section. The first step to using the AUDADC is to set up the audio reference and the Programable Gain Amplifiers (PGAs), and to turn on the microphone bias if an external microphone is connected to an AUDADC input. The following example uses two channels of the PGAs.

```
//
// Power up PrePGA, PGA must be enabled in pairs.
//
am_hal_audadc_refgen_powerup();
am_hal_audadc_pga_powerup(0);
am_hal_audadc_pga_powerup(1);
am_hal_audadc_gain_set(0, 2*PREAMP_FULL_GAIN);
am_hal_audadc_gain_set(1, 2*PREAMP_FULL_GAIN);

//
//  Turn on mic bias
//
am_hal_audadc_micbias_powerup(24);
am_util_delay_ms(400);
```

The next step in initializing the AUDADC is to initialize the instance.

```
//
// Initialize the AUDADC and get the handle.
//
if ( AM_HAL_STATUS_SUCCESS != am_hal_audadc_initialize(0, &g_AUDADCHandle) )
{
    am_util_stdio_printf("Error - reservation of the AUDADC instance
    failed.\n");
}
```

Notice that global variable **g_AUDADCHandle** is used to store a pointer to the handle that is used for all subsequent function access to the ADC instance.

The next step is to power on the AUDADC as follows:

```
//
// Power on the AUDADC.
//
if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_power_control(g_AUDADCHandle,
                                                         AM_HAL_SYSCTRL_WAKE,
                                                         false) )
{
    am_util_stdio_printf("Error - AUDADC power on failed.\n");
}
```

Next, the user should set up the HFRC2 clock to get the precise sample rate.

```
//
// hfrc2 adj.
//
am_hal_mcuctrl_control(AM_HAL_MCUCTRL_CONTROL_EXTCLK32M_KICK_START, false);
am_util_delay_us(1500);
//
// set HF2ADJ for 49.152MHz output
//
am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_HF2ADJ_ENABLE, false);

am_util_delay_us(500);      // wait for adj to apply
```

The AUDADC is configured in two steps. First the AUDADC block is configured, then the internal repeating trigger timer is configured.

```
//
// Set up the AUDADC configuration parameters. These settings are reasonable
// for accurate measurements at a low sample rate.
//
am_hal_audadc_config_t          AUDADCConfig =
{
    .eClock             = AM_HAL_AUDADC_CLKSEL_HFRC2_48MHz,
    .ePolarity          = AM_HAL_AUDADC_TRIGPOL_RISING,
    .eTrigger           = AM_HAL_AUDADC_TRIGSEL_SOFTWARE,
    .eClockMode         = AM_HAL_AUDADC_CLKMODE_LOW_LATENCY,
    .ePowerMode         = AM_HAL_AUDADC_LPMODE1,
    .eRepeat            = AM_HAL_AUDADC_REPEATING_SCAN,
    .eRepeatTrigger     = AM_HAL_AUDADC_RPTTRIGSEL_INT,
    .eSampMode          = AM_HAL_AUDADC_SAMPMODE_MED,
};
```

```
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_configure(g_AUDADCHandle,
        &AUDADCConfig))
    {
        am_util_stdio_printf("Error - configuring AUDADC failed.\n");
    }

    //
    // Set up internal repeat trigger timer
    //
    am_hal_audadc_irtt_config_t      AUDADCIrttConfig =
    {
        .bIrttEnable        = true,
        .eClkDiv            = AM_HAL_AUDADC_RPTT_CLK_DIV32,
        //
        // Adjust sample rate to around 16K.
        // sample rate = eClock/eClkDiv/(ui32IrttCountMax+1)
        //
        .ui32IrttCountMax   = 95,
    };
    am_hal_audadc_configure_irtt(g_AUDADCHandle, &AUDADCIrttConfig);
```

Next, the AUDADC instance is enabled along with the IRTT.

```
    //
    // Enable the AUDADC.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_enable(g_AUDADCHandle))
    {
        am_util_stdio_printf("Error - enabling AUDADC failed.\n");
    }

    //
    // Enable internal repeat trigger timer
    //
    am_hal_audadc_irtt_enable(g_AUDADCHandle);
```

Next, configure the DMA operation and interrupts.

```
    //
    // Configure the AUDADC to use DMA for the sample transfer.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_configure_dma(g_AUDADCHandle,
        &g_sAUDADCDMAConfig))
    {
        am_util_stdio_printf("Error - configuring AUDADC DMA failed.\n");
    }

    //
    // For this example, the samples will be coming in slowly. This means we
    // can afford to wake up for every conversion.
    //
    am_hal_audadc_interrupt_enable(g_AUDADCHandle, AM_HAL_AUDADC_INT_FIFOOVR1 |
AM_HAL_AUDADC_INT_FIFOOVR2 | AM_HAL_AUDADC_INT_DERR | AM_HAL_AUDADC_INT_DCMP );
```

Then set the desired gain configuration.

```
    //
    // Gain setting
    //
    g_sAudadcGainConfig.ui32LGA = (uint32_t)((float)CH_A0_GAIN_DB*2 + 12);
   g_sAudadcGainConfig.ui32HGADELTA = ((uint32_t)((float)CH_A1_GAIN_DB*2 + 12))
- g_sAudadcGainConfig.ui32LGA;
    g_sAudadcGainConfig.eUpdateMode = AM_HAL_AUDADC_GAIN_UPDATE_IMME;
    am_hal_audadc_internal_pga_config(g_AUDADCHandle, &g_sAudadcGainConfig);
```

Next, configure the AUDADC slots for the measurement.

```
     am_hal_audadc_slot_config_t      AUDADCSlotConfig;

    //
    // Set up an AUDADC slot
    //
    AUDADCSlotConfig.eMeasToAvg      = AM_HAL_AUDADC_SLOT_AVG_1;
    AUDADCSlotConfig.ePrecisionMode  = AM_HAL_AUDADC_SLOT_12BIT;
    AUDADCSlotConfig.ui32TrkCyc      = 34;
    AUDADCSlotConfig.eChannel        = AM_HAL_AUDADC_SLOT_CHSEL_SE0;
    AUDADCSlotConfig.bWindowCompare  = false;
    AUDADCSlotConfig.bEnabled        = true;

   if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_configure_slot(g_AUDADCHandle, 0,
       &AUDADCSlotConfig))
    {
        am_util_stdio_printf("Error - configuring AUDADC Slot 0 failed.\n");
    }

    AUDADCSlotConfig.eChannel        = AM_HAL_AUDADC_SLOT_CHSEL_SE1;
   if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_configure_slot(g_AUDADCHandle, 1,
       &AUDADCSlotConfig))
    {
        am_util_stdio_printf("Error - configuring AUDADC Slot 1 failed.\n");
    }
```

Finally, the AUDADC can be triggered manually.

```
    //
    // Trigger the AUDADC sampling for the first time manually.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_sw_trigger(g_AUDADCHandle))
    {
        am_util_stdio_printf("Error - triggering the AUDADC failed.\n");
    }
```

Typically, applications will use an ISR to service subsequent samples once the DMA is complete.

```
void am_audadc0_isr(void)
{
    uint32_t ui32IntMask;
    //
    // Read the interrupt status.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_interrupt_status(g_AUDADCHandle,
        &ui32IntMask, false))
    {
        am_util_stdio_printf("Error reading AUDADC interrupt status\n");
    }

    //
    // Clear the AUDADC interrupt.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_audadc_interrupt_clear(g_AUDADCHandle,
        ui32IntMask))
    {
        am_util_stdio_printf("Error clearing AUDADC interrupt status\n");
    }

    //
    // If we got a DMA complete.
    //
    if (ui32IntMask & AM_HAL_AUDADC_INT_FIFOOVR1)
    {
        if ( AUDADCn(0)->DMASTAT_b.DMACPL )
        {
            am_hal_audadc_interrupt_service(g_AUDADCHandle, &g_sAUDADCDMAConfig);

        }
    }

}
```

3.3 Clock Generator (CLKGEN)

The **CLKGEN** module provides the base clock generation for Apollo4 family SoCs. The HAL provides just three functions

am_hal_clkgen_control() is a general-purpose function to control a variety of clock generation features. Every application needs to choose the clock design and use this function to configure it. See examples below.

```
    //
    // Enable the XT for the RTC.
    //
    am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_XTAL_START, 0);

    //
    // Disable the XTAL.
    //
    am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_RTC_SEL_LFRC, 0);
```

am_hal_clkgen_status_get() provides a way to check on the clock stability after selection.

```
    uint32_t ui32RetVal;
    am_hal_clkgen_status_t sStatus;


    ui32RetVal = am_hal_clkgen_status_get(&sStatus);

    if ( ui32RetVal )
```

```
{
   am_util_stdio_printf("Fail: call to am_hal_clkgen_status() failed, return
   = %d.\n",
                        ui32RetVal);
}

if ( sStatus.ui32SysclkFreq != AM_HAL_CLKGEN_FREQ_MAX_HZ )
{
    am_util_stdio_printf("Fail: ui32SysclkFreq returned %d (expected %d).\n",
                         sStatus.ui32SysclkFreq, AM_HAL_CLKGEN_FREQ_MAX_HZ);
}
```

am_hal_clkgen_clkout_enable provide a way to configure a clock to also be sent to a specific GPIO for this purpose.

```
// Enable CLKOUT on GPIO33 to provide asynchronous interrupt to GPIO13
 am_hal_gpio_pinconfig(GPIO_CLKOUT, g_CLKOUT_33);
 am_hal_clkgen_clkout_enable(true, AM_HAL_CLKGEN_CLKOUT_XTAL_8192);
```

3.4 CM4 Complex Cache/DAXI (CPU)

The Apollo4 family of devices includes a block instance to configure our customization of the interfaces between the Cortex-M4F (CM4) and the peripheral and memory busses. This is contained in the CACHECTRL module HAL, but includes configuration of our Data-AXI (DAXI) cache as well.

3.4.1 Instruction Cache

Most examples initialize the instruction cache to the defined default settings and then enable the cache as follows:

```
//
// Set the default cache configuration
//
am_hal_cachectrl_config(&am_hal_cachectrl_defaults);
am_hal_cachectrl_enable();
```

If required the instruction cache can be disabled as well:

```
//
// Disable the cache
//
am_hal_cachectrl_disable();
```

Also, if required, the cache can be invalidated as follows:

```
//
// Disable and invalidate the cache
//
am_hal_cachectrl_disable();
```

```
am_hal_cachectrl_control(AM_HAL_CACHECTRL_CONTROL_MRAM_CACHE_INVALIDATE, 0);
```

The caching system also provides a way to enable statistical monitoring of the cache for optimization.

```
//
// Now that the config test is done, turn on caching and monitoring.
//
am_hal_cachectrl_config(&am_hal_cachectrl_defaults);
am_hal_cachectrl_enable();
am_hal_cachectrl_control(AM_HAL_CACHECTRL_CONTROL_MONITOR_ENABLE, 0);
```

Finally, there is a function to return the status of the cache

```
am_hal_cachectrl_status_t sStatus;

ui32RetVal = am_hal_cachectrl_status_get(&sStatus);
if ( ui32RetVal )
{
    am_util_stdio_printf("Fail: am_hal_cachectrl_status_get() returned
                         %d).\n",
                         ui32RetVal);
    bTestPass = false;
}
```

3.4.2 Data AXI Cache

The DAXI system is initialized separately from the instruction cache. The functions listed here are contained withing the CACHECTRL HAL for legacy reasons.

Initialization of the default DAXI configuration is performed as follows:

```
uint32_t ui32RetVal;
ui32RetVal = am_hal_daxi_config(&am_hal_daxi_defaults);
if ( ui32RetVal )
{
    am_util_stdio_printf("Fail: am_hal_daxi_config() returned %d).\n",
                         ui32RetVal);
}
```

The application may also set up a custom DAXI configuration as follows:

```
//
//  Set up the DAXICFG.
//
am_hal_daxi_config_t sDAXIConfig =
{
    .bDaxiPassThrough         = false,
    .bAgingSEnabled           = true,
    .eAgingCounter            = AM_HAL_DAXI_CONFIG_AGING_1024,
    .eNumBuf                  = AM_HAL_DAXI_CONFIG_NUMBUF_32,
    .eNumFreeBuf              = AM_HAL_DAXI_CONFIG_NUMFREEBUF_3,
};
am_hal_daxi_config(&sDAXIConfig);
am_hal_delay_us(100);
```

The DAXI also includes a **am_hal_daxi_control()** function that can be used to flush and invalidate the DAXI as follows:

```
am_hal_daxi_control(AM_HAL_DAXI_CONTROL_FLUSH, NULL);
am_hal_daxi_control(AM_HAL_DAXI_CONTROL_INVALIDATE, NULL);
```

The application may also enable/disable the DAXI as follows[1]:

```
am_hal_daxi_control(AM_HAL_DAXI_CONTROL_ENABLE, NULL);
am_hal_daxi_control(AM_HAL_DAXI_CONTROL_DISABLE, NULL);
```

3.5 Inter-IC Sound (I^2S)

The I^2S has four instances, so initialization is as follows:

```
//
// Initialize the I2S and get the handle.
//
if ( am_hal_i2s_initialize(I2S_MODULE_0, &pI2S0Handle)
     != AM_HAL_STATUS_SUCCESS )
{
   am_util_stdio_printf("Error - reservation of the I2S instance failed.\n");
}
```

Notice that global variable **pI2SHandle** is used to store a pointer to the handle that is used for all subsequent function access to the I^2S instance.

The next step is to power on the I^2S as follows:

```
//
// Power on the I2S.
//
if (AM_HAL_STATUS_SUCCESS != am_hal_i2s_power_control(pI2S0Handle,
    AM_HAL_I2S_POWER_ON, false);
{
    am_util_stdio_printf("Error - I2S power on failed.\n");
}
```

The I^2S is configured as follows:

```
static am_hal_i2s_config_t g_sI2S0Config =
{
    .eClock                = eAM_HAL_I2S_CLKSEL_HFRC_6MHz,
    .eDiv3                 = 0,
    .eMode                 = AM_HAL_I2S_IO_MODE_MASTER,
    .eXfer                 = AM_HAL_I2S_XFER_RXTX,
    .eData                 = &g_sI2SDataConfig,
    .eIO                   = &g_sI2SIOConfig
};

if (AM_HAL_STATUS_SUCCESS != am_hal_i2s_configure(pI2S0Handle,
                                                  &g_sI2S0Config))
{
    am_util_stdio_printf("Error - configuring I2S.\n");
}
```

[1] Note that the DAXI configuration has different restrictions based on defects which are explained in the device errata.

The final step for initialization of the I^2S block is to enable the instance as follows:

```
//
// Enable the I2S.
//
if ( am_hal_i2s_enable(pI2S0Handle) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - enabling I2S failed.\n");
}
```

The I^2S block is now ready for operation. The following sequence shows how to enable the HFRC2 using the external 32MHz clock source, then configuring the I^2S DMA and starting a transfer.

```
//
// Enable HFRC2.
//
am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_HFRC2_START, false);
am_util_delay_us(500);      // wait for FLL to lock

//
// Enable EXT CLK32M.
//
if ( (eAM_HAL_I2S_CLKSEL_XTHS_EXTREF_CLK <= g_sI2S0Config.eClock  &&       \
      g_sI2S0Config.eClock <= eAM_HAL_I2S_CLKSEL_XTHS_500KHz) )
{
    am_hal_mcuctrl_control(AM_HAL_MCUCTRL_CONTROL_EXTCLK32M_NORMAL, 0);
    am_util_delay_ms(200);
}

am_hal_i2s_dma_configure(pI2S0Handle, &g_sI2S0Config, &sTransfer0);

//
// Start DMA transaction.
//
am_hal_i2s_dma_transfer_start(pI2S0Handle, &g_sI2S0Config);
```

3.6 General Purpose IO (GPIO)

The GPIO HAL provides functions for establishing individual pin functionality and characteristics. It also provides higher level functions and macros to manipulate pin states in real-time. There are a number of default pin configurations defined. In addition, the GPIO HAL uses **am_hal_pin.h** (an auto-generated file) to define the mapping of pin functionality. Finally, most pin configuration is done inside the Board Support Package (BSP) for the specific evaluation board. This is a set of custom and auto-generated code that this specific to the board. The tools associated with BSP configuration are provided as part of the SDK. See the **am_bsp_pins.src** for more information.

The default pin definitions are of the **am_hal_gpio_pincfg_t** and can be used directly with the **am_hal_gpio_pincconfig()** function. They are:

```
am_hal_gpio_pincfg_default
```

```
am_hal_gpio_pincfg_disabled
am_hal_gpio_pincfg_pulledup_disabled
am_hal_gpio_pincfg_output
am_hal_gpio_pincfg_output_with_read
am_hal_gpio_pincfg_input
am_hal_gpio_pincfg_tristate
am_hal_gpio_pincfg_opendrain
```

The predefined pin manipulation macros are:

```
am_hal_gpio_input_read(n)
am_hal_gpio_output_read(n)
am_hal_gpio_enable_read(n)
am_hal_gpio_output_clear(n)
am_hal_gpio_output_set(n)
am_hal_gpio_output_toggle(n)
am_hal_gpio_output_tristate_disable(n)
am_hal_gpio_output_tristate_enable(n)
am_hal_gpio_output_tristate_toggle(n)
am_hal_gpio_intdir_toggle(n)
```

There are also a series of functions to manipulate pins. The first are to configure the pins. **am_hal_gpio_pinconfig()** is the primary function used to set up the pin configuration.

```
//
// Set a pin to act as our ADC input
//
am_hal_gpio_pinconfig(19, g_AM_PIN_19_ADCSE0);
```

am_hal_gpio_pinconfig_get() returns the current configuration of a pin.

The next set of functions are used to read and write the current state of the pins. **am_hal_gpio_state_read()** returns the pin input, output, and/or output enable values.

```
//
// Check to see if the LED pin is enabled.
//
ui32Ret = am_hal_gpio_state_read(psLEDs[ui32LEDNum].ui32GPIONumber,
    AM_HAL_GPIO_ENABLE_READ, &ui32Value);
```

am_hal_gpio_state_write() sets the pin output drive, and/or output enable values

```
//
// If it was enabled, turn if off.
//
am_hal_gpio_state_write(psLEDs[ui32LEDNum].ui32GPIONumber,
                    AM_HAL_GPIO_OUTPUT_TRISTATE_DISABLE);
```

The final set of functions are used to control and service pin interrupts:

am_hal_gpio_interrupt_control() enables one or more interrupts for the pin.

```
//
// Enable the GPIO/button interrupt.
```

```
//
am_hal_gpio_interrupt_control(AM_HAL_GPIO_INT_CHANNEL_0,
    AM_HAL_GPIO_INT_CTRL_INDV_ENABLE,
    (void *)&ui32BUTTON0GpioNum);
```

am_hal_gpio_interrupt_clear() clears the specified interrupts.

```
// Set up the host IO interrupt
am_hal_gpio_interrupt_clear(AM_HAL_GPIO_INT_CHANNEL_0,
    (am_hal_gpio_mask_t*)&IntNum);
```

am_hal_gpio_interrupt_irq_status_get() reads the interrupt status for a given pin interrupt request and **am_hal_gpio_interrupt_irq_clear()** clears a given pin interrupt request.

```
am_hal_gpio_interrupt_irq_status_get(GPIO0_001F_IRQn, false, &ui32IntStatus);
am_hal_gpio_interrupt_irq_clear(GPIO0_001F_IRQn, ui32IntStatus);
```

am_hal_gpio_interrupt_register() registers an interrupt handler function to called for a given pin.

```
// Register handler for IOS => IOM interrupt
am_hal_gpio_interrupt_register(AM_HAL_GPIO_INT_CHANNEL_0,
    HANDSHAKE_PIN,(am_hal_gpio_handler_t)hostint_handler, NULL);
```

am_hal_gpio_interrupt_service() is called to service registered pin handlers.

```
am_hal_gpio_interrupt_service(AM_COOPER_IRQn, ui32IntStatus);
```

3.7 Graphics Processing Unit (GPU)

The NemaGFX SDK provides a rich set of primitives for a variety of graphics operations. See the *NemaGFX Programming Guide* for more information. This section merely explains the basics of the GPU subsystem initialization and operation.

Initialization of the GPU subsystems starts with the **nema_init()**. This function initializes the GPU and calls the **nema_sys_init()** function which is responsible for the graphics subsystem initialization. The graphics subsystem includes:

- Graphics memory pool initialization
- GPU Interrupt synchronization mutex initialization
- Ring-buffer allocation and initialization

Applications may use the default setting, or adjust them based on the system's overall needs. See **nema_hal.c**.

After GPU initialization, a Command List (CL) can be created and used to submit tasks to GPU. A typical routine for drawing with CL would be the following:

```
nema_cmdlist_t cl = nema_cl_create ();// Create a new CL
nema_cl_bind(&cl);// Bind it
/* Drawing Operations */ // Draw scene
nema_cl_unbind();// Unbind CL (optionally)
nema_cl_submit(&cl);// Submit CL for execution
```

Every drawing operation should have an effect on a given destination texture. The texture must reside in some memory space which is visible to the GPU. The **nema_bind_dst_tex()** function binds a texture to serve as destination.

```
nema_bind_dst_tex(cur_fb_base_phys, fb[0].w, fb[0].h, fb[0].format, -1);
```

The texture's attributes (GPU address, width, height, format and stride) are written inside the bound CL. Each subsequent drawing operation will have an effect on this destination texture. Most common graphics operations include some kind of image blitting (copying), such as drawing a back-ground image, drawing GUI icons, or even font rendering. The **nema_bind_src_tex()** function binds a texture to be used as foreground:

```
nema_bind_src_tex(image_assets[img_asset_id]->bo.base_phys,
                  image_assets[img_asset_id]->w,
    image_assets[img_asset_id]->h,
    image_assets[img_asset_id]->format, image_assets[img_asset_id]->stride,
    image_assets[img_asset_id]->sampling_mode);
```

When drawing a scene, it is often necessary to be able to define a rectangular area that the GPU is allowed to draw. This way, if some parts of a primitive (e.g., a triangle) falls outside the clipping area, that part is not going to be drawn at all, assuring correctness, better performance and improved power efficiency. The Clipping Rectangle can be defined by the **nema_set_clip()** function as follows:

```
nema_set_clip (0, 0, RESX, RESY);
```

This function defines a Clipping Rectangle whose upper left vertex coordinates are (x = 0, y = 0) and its dimensions are (w = RESX by h = RESY).

When building a graphical interface, the developer has to define what would be the result of drawing a pixel on the canvas. Since the canvas already contains the previous drawn scene, there must be a consistent way to determine how the source or foreground color (the one that is going to be drawn) will blend with the destination or background color that is already drawn. The source pixel can be fully opaque, thus will be drawn over the destination one, or it can be translucent and the result would be a blend of both the source and destination colors. **nema_set_blend_fill()** function refers to blending when filling a primitive (e.g., triangle) with a color as follows:

```
nema_set_blend_fill(NEMA_BL_SIMPLE);
```

nema_set_blend_blit() function refers to blitting a texture.

```
nema_set_blend_blit(NEMA_BL_SRC);
```

After setting up the above, the CL contains all the information needed to blit an image or fill a geometric primitive with color.

Typically, the GPU will raise an interrupt when it has finished executing a Command List. The ID of the last Command List that has been executed can be read from the Configuration Register **NEMA_CLID**. The **nema_wait_irq_cl()** function should be called wait until the content of the **NEMA_CLID** register is greater or equal than the **cl_id** argument.

```
void nema_wait_irq_cl(intcl_id)
{
    while( nema_reg_read(NEMA_CLID) < cl_id)
    {
        nema_wait_irq ();
    }
}
```

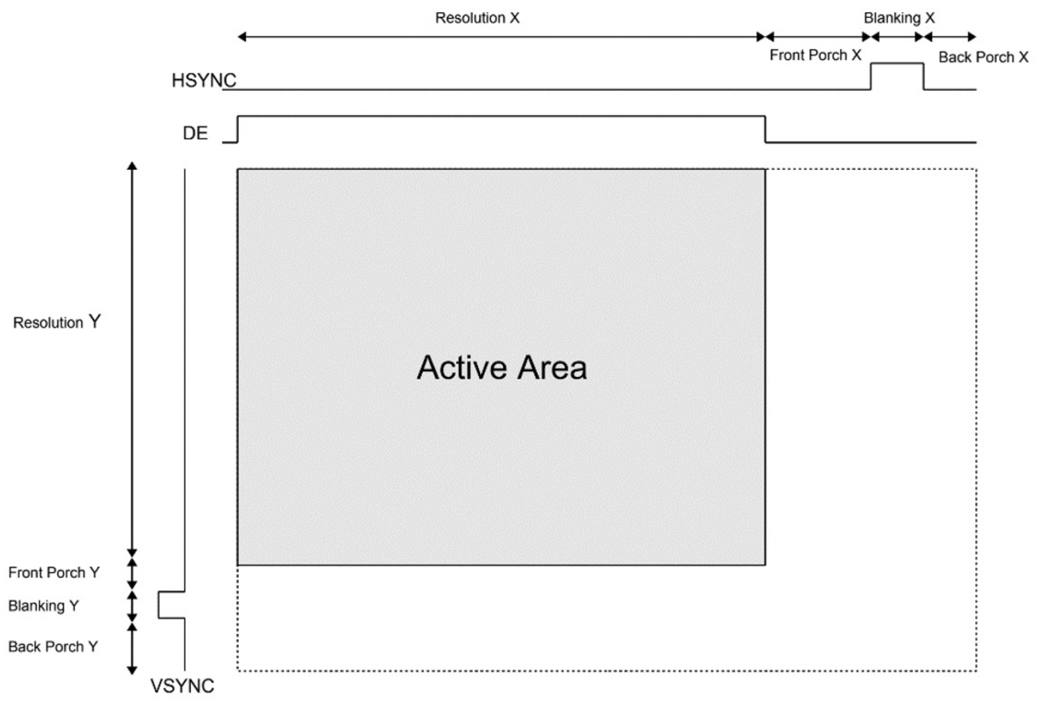
3.8 Display Controller (DC)

Like the GPU, the DC is initialized by using the NemaGFX SDK functions as well. Initialization includes enabling (powering-up) the DC if it is not already on, then calling **nemadc_init()** function.

```
int32_t i32Ret = 1;
bool status;
am_hal_pwrctrl_periph_enabled(AM_HAL_PWRCTRL_PERIPH_DISP, &status);
if ( !status )
{
    am_hal_pwrctrl_periph_enable(AM_HAL_PWRCTRL_PERIPH_DISP);
    i32Ret = nemadc_init();
}
```

After basic initialization, the timing parameters must be set correctly.

Figure 3-1: Timing Parameters



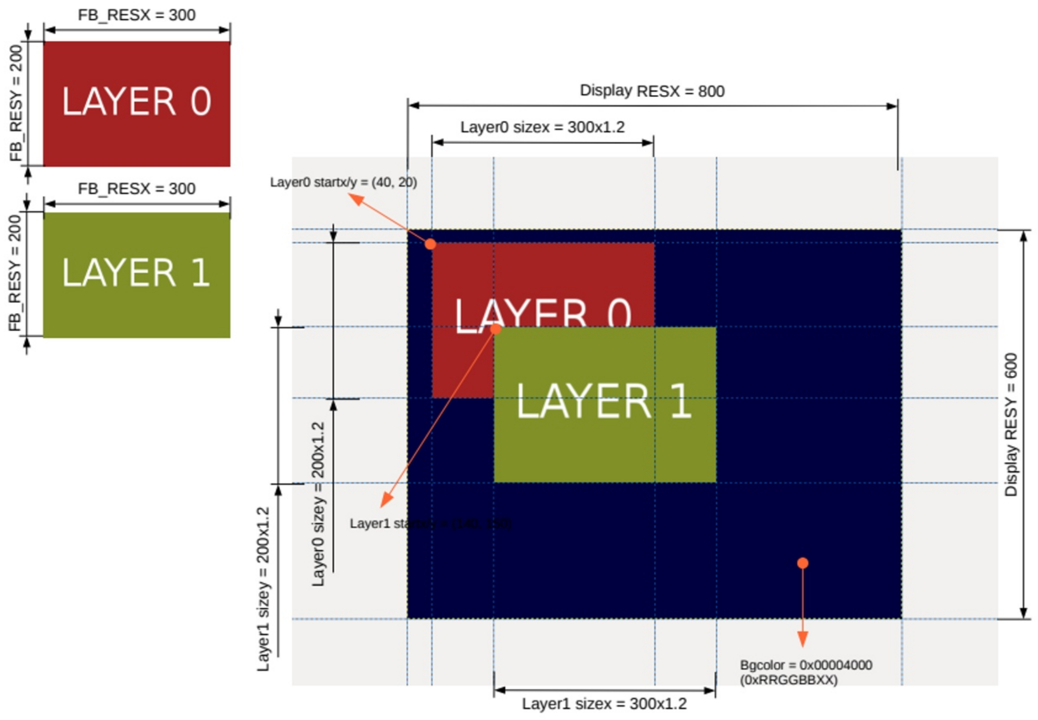
These parameters include setting display resolution and blanking the front and back porch.

```
    nemadc_timing(i32ResX, 4, 10, 10, i32ResY, 10, 50, 10);
```

DC supports up to four layers. Layers are visual elements that are stacked to compose the final displayed image. They are completely independent to each other, therefore must be set separately.

In order to set a layer, a **nemadc_layer_t struct** containing layer's info must be initialized. This struct type contains information regarding the layer, such as resolution, stride, format, etc.

Figure 3-2: Struct Type Information



Layer's startx/starty set layer’s top left corner position, (0, 0) is on the top left corner of the display;

Layer’s resx/resy correspond to the original framebuffer’s resolution;

Layer’s sizex/sizey correspond to final scaled dimensions in pixels.

Layers are set as follows:

```
    nemadc_set_layer(0, &layer[cur_fb]);
```

There is also the possibility of setting only the layer’s physical address. This is useful in case the developer wants to swap buffers:

```
    nemadc_set_layer(0, &layer[cur_fb]);
```

After setting layers, the following function launches the frame transfer.

```
    nemadc_set_mode(0);
```

Besides the above, DC implements a programmable global and per-layer Gamma Look Up Table (LUT) for gamma correction. Gamma LUT consists of a 3x256x8 memory array that holds the RGB values for each of the 256 colors in the palette. In order to program it, Our SDK implements get and set calls for both the global and the per layer Gamma LUT.

```
int nemadc_get_palette ( int index )
void nemadc_set_palette ( int index , int color )
int nemadc_get_layer_gamma_lut ( int layer_no , int index )
void nemadc_set_layer_gamma_lut ( int layer_no , int index , int color )
```

While DC is busy scanning the current frame from memory, the frame should never be altered. Any modifications to the layer information or address should be done once DC is done with the frame refresh. To achieve this, our SDK implements a call that waits for Vertical Synchronization (VSync)

```
// Wait for transfer to be completed
nemadc_wait_vsync();
```

Its purpose is to yield the CPU (suspend the core or current RTOS task) until the DC finishes its current refresh cycle, continuing the execution when done, in order to avoid screen tearing or visual artifacts. The execution is resumed by DC's interrupt. For more details, refer to the default implementation in **nema_dc_hal.c**

3.9 SPI/I^{2}C I/O Master (IOM)

The IOM has 8 instances[1], so initialization is as follows:

```
//
// Initialize the IOM.
//
if ( am_hal_iom_initialize(iomModule, &g_IOMHandle) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - reservation of the IOM instance %d failed.\n",
                         iomModule);
}
```

Notice that global variable **g_IOMHandle** is used to store a pointer to the handle that is used for all subsequent function access to the IOM instance.

The next step is to power on the IOM domain as follows:

```
//
// Power-on the IOM.
//
if ( am_hal_iom_power_ctrl(g_IOMHandle,
                           AM_HAL_SYSCTRL_WAKE,
                           false) != AM_HAL_STATUS_SUCCESS)
{
    am_util_stdio_printf("Error - IOM power-on failed.\n");
}
```

Each IOM instance can be configured as either SPI or I^{2}C as follows:

```
//
// Set the required configuration settings for the IOM for SPI mode.
//
static am_hal_iom_config_t     g_sIOMSpiConfig =
{
    .eInterfaceMode       = AM_HAL_IOM_SPI_MODE,
    .ui32ClockFreq        = AM_HAL_IOM_1MHZ,
    .eSpiMode             = AM_HAL_IOM_SPI_MODE_0,
    .ui32NBTxnBufLength   = 0,
    .pNBTxnBuf = NULL,
```

[1] Note that on the Apollo4 Blue devices IOM4 is internal and dedicated to BLE control, so this instance is not available for general use.

```
};
if ( am_hal_iom_configure(g_IOMHandle, &g_sIOMSpiConfig)
                          != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - IOM SPI configuration failed.\n");
}
//
// Set the required configuration settings for the IOM for I2C mode.
//
static am_hal_iom_config_t     g_sIOMI2cConfig =
{
    .eInterfaceMode       = AM_HAL_IOM_I2C_MODE,
    .ui32ClockFreq        = AM_HAL_IOM_1MHZ,
    .eSpiMode             = AM_HAL_IOM_SPI_MODE_0,
    .ui32NBTxnBufLength   = 0,
    .pNBTxnBuf = NULL,
};

if ( am_hal_iom_configure(g_IOMHandle, &g_sIOMI2cConfig)
     != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - IOM I2C configuration failed.\n");
}
```

The final step for initialization of the IOM instance is to enable the instance as follows:

```
//
// Enable the IOM.
//
If ( am_hal_iom_enable(g_IOMHandle) != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - IOM enable failed.\n");
}
```

At this point, the IOM is operational and can accepted blocking transfer as follows:

```
am_hal_iom_transfer_t Transaction;

//
// Create the transaction.
//
Transaction.ui32InstrLen    = ui32InstrLen;
Transaction.ui64Instr       = ui64Instr;
Transaction.eDirection      = AM_HAL_IOM_TX;
Transaction.ui32NumBytes    = ui32NumBytes;
Transaction.pui32TxBuffer   = pData;
Transaction.uPeerInfo.ui32I2CDevAddr = ui8DevAddr;
Transaction.bContinue       = bCont;
Transaction.ui8RepeatCount  = 0;
Transaction.ui32PauseCondition = 0;
Transaction.ui32StatusSetClr = 0;

//
// Execute the transction over IOM.
//
if (am_hal_iom_blocking_transfer(g_IOMHandle, &Transaction))
{
    return AM_HAL_STATUS_ERROR;
}
```

A blocking transfer will return from the function call when the entire transfer is complete.

In addition, the IOM can accept non-blocking transfers as follows:

```
    am_hal_iom_transfer_t Transaction;
    //
    // Set up the IOM transaction to write the offset (address) and data.
    //
    Transaction.eDirection      = AM_HAL_IOM_TX;
    Transaction.ui32InstrLen    = 3;
    Transaction.ui64Instr       = ui32WriteAddress & 0x00FFFFFF;
    Transaction.ui32NumBytes    = ui32NumBytes;
    Transaction.pui32TxBuffer   = (uint32_t *)pui8TxBuffer;
    Transaction.bContinue       = false;

    //
    // Start the transaction
    //
    if ( am_hal_iom_nonblocking_transfer(pIom->pIomHandle, &Transaction,
         pfnCallback, pCallbackCtxt) != AM_HAL_STATUS_SUCCESS )
        {
            return AM_DEVICES_MB85RS1MT_STATUS_ERROR;
        }
```

In this particular transfer, pointers to a callback function and context are passed. This indicates that this is the last of series of transfers. When complete, the IOM HAL will execute the callback function. The non-blocking transfers execute asynchronously and notify the application through this callback mechanism and interrupts.

The IOM ISR should look as follows:

```
void psram_iom_isr(void)
{
    uint32_t ui32Status;

    if (!am_hal_iom_interrupt_status_get(g_pIOMHandle, true, &ui32Status))
    {
        if ( ui32Status )
        {
            am_hal_iom_interrupt_clear(g_pIOMHandle, ui32Status);
            am_hal_iom_interrupt_service(g_pIOMHandle, ui32Status);
        }
    }
}
```

The ISR reads the current interrupt status, then clears it and executes the interrupt service routine. This is the context where IOM non-blocking callbacks are executed.

3.10 SPI/I^{2}C I/O Slave (IOS)

The IOS has a single instance, so initialization is as follows:

```
//
// Initialize the IOS.
//
if ( am_hal_ios_initialize(0, &g_pIOSHandle) != AM_HAL_STATUS_SUCCESS )
{
   am_util_stdio_printf("Error - reservation of the IOS instance failed.\n");
}
```

Notice that global variable **g_pIOSHandle** is used to store a pointer to the handle that is used for all subsequent function access to the IOS instance.

The next step is to power on the IOS domain as follows:

```
//
// Power-on the IOS.
//
if ( am_hal_ios_power_ctrl(g_pIOSHandle,
                           AM_HAL_SYSCTRL_WAKE,
                           false) != AM_HAL_STATUS_SUCCESS)
{
    am_util_stdio_printf("Error - IOS power-on failed.\n");
}
```

The IOS instance can be configured as either SPI or I^{2}C as follows:

```
//
// Set the required configuration settings for the IOS for SPI mode.
//
static am_hal_ios_config_t g_sIOSSpiConfig =
{
    // Configure the IOS in SPI mode.
    .ui32InterfaceSelect = AM_HAL_IOS_USE_SPI,

    // Eliminate the "read-only" section, so an external host can use the
    // entire "direct write" section.
    .ui32ROBase = 0x78,

    // Making the "FIFO" section as big as possible.
    .ui32FIFOBase = 0x80,

    // We don't need any RAM space, so extend the FIFO all the way to the end
    // of the LRAM.
    .ui32RAMBase = 0x100,

    // FIFO Threshold - set to half the size
    .ui32FIFOThreshold = 0x20,

    .pui8SRAMBuffer = g_pui8TxFifoBuffer,
    .ui32SRAMBufferCap = AM_IOS_TX_BUFSIZE_MAX,
};

if ( am_hal_ios_configure(g_pIOSHandle, &g_sIOSSpiConfig)
     != AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error - IOS SPI configuration failed.\n");
    }
    //
    // Set the required configuration settings for the IOS for I2C mode.
    //
    am_hal_ios_config_t g_sIOSI2cConfig =
    {
        // Configure the IOS in I2C mode.
        .ui32InterfaceSelect = AM_HAL_IOS_USE_I2C |
                               AM_HAL_IOS_I2C_ADDRESS(I2C_ADDR << 1),

        // Eliminate the "read-only" section, so an external host can use the
        // entire "direct write" section.
        .ui32ROBase = 0x78,

        // Set the FIFO base to the maximum value, making the "direct write"
        // section as big as possible.
        .ui32FIFOBase = 0x80,

        // We don't need any RAM space, so extend the FIFO all the way to the end
        // of the LRAM.
        .ui32RAMBase = 0x100,

        // FIFO Threshold - set to half the size
        .ui32FIFOThreshold = 0x40,

        .pui8SRAMBuffer = g_pui8TxFifoBuffer,
        .ui32SRAMBufferCap = AM_IOS_TX_BUFSIZE_MAX,
    };
    if ( am_hal_ios_configure(g_pIOSHandle, &g_sIOSI2cConfig != AM_HAL_STATUS_-
SUCCESS )
    {
        am_util_stdio_printf("Error - IOM I2C configuration failed.\n");
    }
```

For the IOS, the enable/disable functions are called from within the configure function.

At this point, the IOS is operational and can accept functions to write to the IOS FIFO:

```
am_hal_ios_fifo_write(g_pIOSHandle, &g_pui8TestBuf[g_sendIdx], SENSOR0_DATA_-
SIZE, &numWritten);
```

The IOS ISR should read the interrupt status, clear it and call the service routine. Depending upon the application the ISR may also intercept error indications and in-band commands from the host.

```
void am_ioslave_ios_isr (void)
{
    uint32_t ui32Status;
    am_hal_ios_interrupt_status_get(g_pIOSHandle, false, &ui32Status))
    am_hal_iom_interrupt_clear(g_pIOMHandle, ui32Status);
    if (ui32Status & AM_HAL_IOS_INT_FSIZE)
    {
        //
        // Service the IOS FIFO if necessary.
        //
        am_hal_ios_interrupt_service(g_pIOSHandle, ui32Status);
    }
}
```

3.11 Multibit SPI Master (MSPI)

The MSPI has 3 instances, so initialization is as follows:

```
    //
    // Initialize the MSPI.
    //
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_initialize(ui32Module, &g_MSPIHan-
dle))
    {
        am_util_stdio_printf("Error - Failed to initialize MSPI.\n");
    }
```

Notice that global variable **g_MSPIHandle** is used to store a pointer to the handle that is used for all subsequent function access to the MSPI instance.

The next step is to power on the MSPI domain as follows:

```
    //
    // Power-on the MSPI.
    //
    if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_power_control(g_MSPIHandle,
                                                           AM_HAL_SYSCTRL_WAKE,
                                                           false))
    {
        am_util_stdio_printf("Error - Failed to power on MSPI.\n");
    }
```

Each MSPI instance can be configured in two steps. First the base MSPI instance is configured. This includes the buffer for Command Queue/DMA transactions.

```
    //
    // Set the required configuration settings for the MSPI.
    //
    am_hal_mspi_config_t mspiCfg =
    {
        .ui32TCBSize          = ui32NBTxnBufLength,
        .pTCB                 = pNBTxnBuf,
        .bClkonD4             = 0
    };
    if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_configure(g_MSPIHandle, &mspiCfg))
    {
        am_util_stdio_printf("Error - Failed to configure MSPI device.\n");
    }
```

Next, the MSPI is configured for the specific device. This configuration can vary widely depending on the end device. The customer can look at our device driver examples for MSPI to find more details. Configuration for a Quad PSRAM follows:

```
    am_hal_mspi_dev_config_t  MSPIDevConfig =
    {
        .ui8TurnAround        = 8,
        .eAddrCfg             = AM_HAL_MSPI_ADDR_3_BYTE,
        .eInstrCfg            = AM_HAL_MSPI_INSTR_1_BYTE,
        .ui16ReadInstr        = AM_DEVICES_MSPI_PSRAM_FAST_READ,
        .ui16WriteInstr       = AM_DEVICES_MSPI_PSRAM_WRITE,
        .eDeviceConfig        = AM_HAL_MSPI_FLASH_SERIAL_CE1,
        .eSpiMode             = AM_HAL_MSPI_SPI_MODE_0,
        .eClockFreq           = AM_HAL_MSPI_CLK_16MHZ,
        .bSendAddr            = true,
        .bSendInstr           = true,
        .bTurnaround          = true,
        .ui8WriteLatency      = 0,
        .bEnWriteLatency      = false,
        .bEmulateDDR          = false,
        .ui16DMATimeLimit     = 80,
        .eDMABoundary         = AM_HAL_MSPI_BOUNDARY_BREAK1K,
    };
    if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_device_configure(g_MSPIHandle, &
        MSPIDevConfig))
    {
        am_util_stdio_printf("Error - Failed to configure MSPI device.\n");
        return AM_DEVICES_MSPI_PSRAM_STATUS_ERROR;
    }
```

The final step for initialization of the MSPI instance is to enable the instance as follows:

```
//
// Enable the MSPI.
//
if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_enable(g_MSPIHandle))
{
    am_util_stdio_printf("Error - Failed to enable MSPI.\n");
}
```

At this point, the MSPI is operational and can accepted blocking transfer as follows:

```
am_hal_mspi_pio_transfer_t  Transaction;

// Create the individual write transaction.
Transaction.ui32NumBytes           = ui32NumBytes;
Transaction.bScrambling            = false;
Transaction.eDirection             = AM_HAL_MSPI_TX;
Transaction.bSendAddr              = bSendAddr;
Transaction.ui32DeviceAddr         = ui32Addr;
Transaction.bSendInstr             = true;
Transaction.ui16DeviceInstr        = ui8Instr;
Transaction.bTurnaround            = false;
Transaction.bQuadCmd               = false;
Transaction.pui32Buffer            = pData;

// Execute the transction over MSPI.
if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_blocking_transfer(g_MSPIHandle,
                                   &Transaction,
                                   AM_DEVICES_MSPI_PSRAM_TIMEOUT))
{
   am_util_stdio_printf("Error - Failed to complete the blocking transfer to
   MSPI.\n");
}
```

A blocking transfer will return from the function call when the entire transfer is complete.

In addition, the MSPI can accept non-blocking transfers as follows:

```
am_hal_mspi_dma_transfer_t    Transaction;
```

```
    // Set the DMA priority
    Transaction.ui8Priority = 1;
    Transaction.eDirection = AM_HAL_MSPI_TX;
    Transaction.ui32TransferCount = size;
    Transaction.ui32DeviceAddress = ui32Address;
    Transaction.ui32SRAMAddress = (uint32_t)pui8Buffer;

    if (AM_HAL_STATUS_SUCCESS != am_hal_mspi_nonblocking_transfer(pPsram->pMspi-
Handle,
                                           &Transaction,
                                           AM_HAL_MSPI_TRANS_DMA,
                                           pfnCallback,
                                           pCallbackCtxt))
    {
      am_util_stdio_printf("Error - Failed to complete the non-blocking transfer
      to MSPI.\n");
    }
```

In this particular transfer, pointers to a callback function and context are passed. This indicates that this is the last of series of transfers. When complete, the MSPI HAL will execute the callback function. The non-blocking transfers execute asynchronously and notify the application through this callback mechanism and interrupts.

The MSPI ISR should look as follows:

```
void mspi_isr(void)
{
    uint32_t      ui32Status;

    am_hal_mspi_interrupt_status_get(g_pHandle, &ui32Status, false);
    am_hal_mspi_interrupt_clear(g_pHandle, ui32Status);
    am_hal_mspi_interrupt_service(g_pHandle, ui32Status);
}
```

The ISR reads the current interrupt status, then clears it and executes the interrupt service routine. This is the context where MSPI non-blocking callbacks are executed.

3.12 Pulse Density Modulation (PDM)

The PDM has four instances, so initialization is as follows:

```
    //
    // Initialize, power-up, and configure the PDM.
    //
    am_hal_pdm_initialize(PDM_MODULE, &PDMHandle);
```

Notice that global variable **PDMHandle** is used to store a pointer to the handle that is used for all subsequent function access to the PDM instance.

The next step is to power on the PDM as follows:

```
    am_hal_pdm_power_control(PDMHandle, AM_HAL_PDM_POWER_ON, false);
```

The PDM block operates with a high frequency clock source similar to I^{2}S. The following sequence shows how to enable the HFRC2 using the external 32MHz clock source:

```
//
// use external XTHS, not reference clock
//
am_hal_mcuctrl_control(AM_HAL_MCUCTRL_CONTROL_EXTCLK32M_KICK_START, false);

//
// enable HFRC2
//
am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_HFRC2_START, false);
am_util_delay_us(200);      // wait for FLL to lock

//
// set HF2ADJ for 24.576MHz output
//
am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_HF2ADJ_ENABLE, false);
am_util_delay_us(500);      // wait for adj to apply
```

The PDM is configured as follows:

```
am_hal_pdm_config_t g_sPdmConfig =
{
    //
    // Example setting:
    //  1.5MHz PDM CLK OUT:
    //      AM_HAL_PDM_CLK_HFRC2ADJ_24_576MHZ, AM_HAL_PDM_MCLKDIV_1,
    //      AM_HAL_PDM_PDMA_CLKO_DIV7
    //  48.00KHz 24bit Sampling:
    //      DecimationRate = 16
    //
    .ePDMClkSpeed = AM_HAL_PDM_CLK_HFRC2ADJ_24_576MHZ,
    .eClkDivider = AM_HAL_PDM_MCLKDIV_1,
    .ePDMAClkOutDivder = AM_HAL_PDM_PDMA_CLKO_DIV7,
    .ui32DecimationRate = 16,
    .eLeftGain = AM_HAL_PDM_GAIN_0DB,
    .eRightGain = AM_HAL_PDM_GAIN_0DB,
    .eStepSize = AM_HAL_PDM_GAIN_STEP_0_13DB,
    .bHighPassEnable = AM_HAL_PDM_HIGH_PASS_ENABLE,
    .ui32HighPassCutoff = 0x3,
    .bDataPacking = 1,
    .ePCMChannels = AM_HAL_PDM_CHANNEL_LEFT,
    .bPDMSampleDelay = AM_HAL_PDM_CLKOUT_PHSDLY_NONE,
    .ui32GainChangeDelay = AM_HAL_PDM_CLKOUT_DELAY_NONE,
    .bSoftMute = 0,
    .bLRSwap = 0,
};

am_hal_pdm_configure(PDMHandle, &g_sPdmConfig);
```

Next, the application must set up the FIFO threshold:

```
am_hal_pdm_fifo_threshold_setup(PDMHandle, FIFO_THRESHOLD_CNT);
```

The final step for initialization of the PDM block is to enable the instance as follows:

```
    //
    // Start the data transfer.
    //
    am_hal_pdm_enable(PDMHandle);
```

The PDM block is now ready for operation. The application can start the PDM DMA transfer.

```
    am_hal_pdm_transfer_t sTransfer =
    {
        .ui32TargetAddr = ui32PDMDataPtr;
        .ui32TargetAddrReverse = sTransfer.ui32TargetAddr +
        sTransfer.ui32TotalCount;
        .ui32TotalCount        = DMA_BYTES,
    };

    //
    // Start data conversion
    //
    am_hal_pdm_dma_start(PDMHandle, &sTransfer);
```

3.13 Power Control (PWRCTRL)

The Power Control HAL does not conform to the uniform driver API design. This is mostly because the PWRCTRL HAL is designed to provide a number of different configuration functions which configure a number of different aspects of the design. The following sections explain these functions in more detail.

3.13.1 MCU Performance

The PWRCTRL HAL provides two functions to adjust and check the main CPU performance mode. The first allows the customer to select either Low-Power mode (96MHz operation) or High-Performance mode (192MHz operation).

```
if (am_hal_pwrctrl_mcu_mode_select(AM_HAL_PWRCTRL_MCU_MODE_LOW_POWER) !=
AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error switching to low power mode\n);
}
if (am_hal_pwrctrl_mcu_mode_select(AM_HAL_PWRCTRL_MCU_MODE_HIGH_PERFORMANCE) !=
AM_HAL_STATUS_SUCCESS )
{
    am_util_stdio_printf("Error switching to high performance mode\n);
}
```

There is also a function read the current power mode of the CPU.

```
if ( am_hal_pwrctrl_mcu_mode_status(&eCurrentPowerMode)!= AM_HAL_STATUS_SUCCESS
)
{
    am_util_stdio_printf("Error reading current CPU power mode status\n);
}
```

3.13.2 DTCM, NVM, SSRAM and Extended SRAM configuration and retention

The memory is configured by three separate function. First the main MCU memory, which includes DTCM and Cache is configure.

```
const am_hal_pwrctrl_mcu_memory_config_t    g_DefaultMcuMemCfg =
{
    .eCacheCfg          = AM_HAL_PWRCTRL_CACHE_ALL,
    .bRetainCache       = true,
    .eDTCMCfg           = AM_HAL_PWRCTRL_DTCM_384K,
    .eRetainDTCM        = AM_HAL_PWRCTRL_DTCM_384K,
    .bEnableNVM0        = true,
    .bRetainNVM0        = false
};
am_hal_pwrctrl_mcu_memory_config((am_hal_pwrctrl_mcu_memory_config_t *)&g_De-
faultMcuMemCfg);
```

Then the Shared SRAM (SSRAM) is configured as follows:

```
const am_hal_pwrctrl_sram_memcfg_t          g_DefaultSRAMCfg =
{
    .eSRAMCfg           = AM_HAL_PWRCTRL_SRAM_ALL,
    .eActiveWithMCU     = AM_HAL_PWRCTRL_SRAM_NONE,
    .eActiveWithGFX     = AM_HAL_PWRCTRL_SRAM_NONE,
    .eActiveWithDISP    = AM_HAL_PWRCTRL_SRAM_NONE,
    .eActiveWithDSP     = AM_HAL_PWRCTRL_SRAM_NONE,
    .eSRAMRetain        = AM_HAL_PWRCTRL_SRAM_ALL
};
am_hal_pwrctrl_sram_config((am_hal_pwrctrl_sram_memcfg_t *)&g_DefaultSRAMCfg);
```

Finally, if Extended SRAM (a.k.a. DSP RAM) is used, then it may also be configured.

```
const am_hal_pwrctrl_dsp_memory_config_t    g_DefaultDSPMemCfg =
{
    .bEnableICache      = false,
    .bRetainCache       = false,
    .bEnableRAM         = true,
    .bActiveRAM         = false,
    .bRetainRAM         = true
};
am_hal_pwrctrl_dsp_memory_config(AM_HAL_DSP0, & g_DefaultDSPMemCfg);
am_hal_pwrctrl_dsp_memory_config(AM_HAL_DSP1, & g_DefaultDSPMemCfg);
```

Each of these functions has **corresponding _get** function which allows the user to query the current configuration and then make only incremental changes.

3.13.3 Peripheral power domain control

Control of the peripheral power domains is done via the peripheral enable/disable functions. The set of peripheral domains that can be power controlled is list below. The user should be aware that some of these domains are tied together. For example, enabling IOM0 will also enable IOM1-3 as they lie within the same power domain.

```
typedef enum
{
    AM_HAL_PWRCTRL_PERIPH_IOS,
    AM_HAL_PWRCTRL_PERIPH_IOM0,
    AM_HAL_PWRCTRL_PERIPH_IOM1,
    AM_HAL_PWRCTRL_PERIPH_IOM2,
    AM_HAL_PWRCTRL_PERIPH_IOM3,
    AM_HAL_PWRCTRL_PERIPH_IOM4,
    AM_HAL_PWRCTRL_PERIPH_IOM5,
    AM_HAL_PWRCTRL_PERIPH_IOM6,
    AM_HAL_PWRCTRL_PERIPH_IOM7,
    AM_HAL_PWRCTRL_PERIPH_UART0,
    AM_HAL_PWRCTRL_PERIPH_UART1,
    AM_HAL_PWRCTRL_PERIPH_UART2,
    AM_HAL_PWRCTRL_PERIPH_UART3,
    AM_HAL_PWRCTRL_PERIPH_ADC,
    AM_HAL_PWRCTRL_PERIPH_MSPI0,
    AM_HAL_PWRCTRL_PERIPH_MSPI1,
    AM_HAL_PWRCTRL_PERIPH_MSPI2,
    AM_HAL_PWRCTRL_PERIPH_GFX,
    AM_HAL_PWRCTRL_PERIPH_DISP,
    AM_HAL_PWRCTRL_PERIPH_DISPPHY,
    AM_HAL_PWRCTRL_PERIPH_CRYPTO,
    AM_HAL_PWRCTRL_PERIPH_SDIO,
    AM_HAL_PWRCTRL_PERIPH_USB,
    AM_HAL_PWRCTRL_PERIPH_USBPHY,
    AM_HAL_PWRCTRL_PERIPH_DEBUG,
    AM_HAL_PWRCTRL_PERIPH_AUDREC,
    AM_HAL_PWRCTRL_PERIPH_AUDPB,
    AM_HAL_PWRCTRL_PERIPH_PDM0,
    AM_HAL_PWRCTRL_PERIPH_PDM1,
    AM_HAL_PWRCTRL_PERIPH_PDM2,
    AM_HAL_PWRCTRL_PERIPH_PDM3,
    AM_HAL_PWRCTRL_PERIPH_I2S0,
    AM_HAL_PWRCTRL_PERIPH_I2S1,
    AM_HAL_PWRCTRL_PERIPH_AUDADC,
    AM_HAL_PWRCTRL_PERIPH_MAX
} am_hal_pwrctrl_periph_e;

if ( am_hal_pwrctrl_periph_enable(AM_HAL_PWRCTRL_PERIPH_SDIO)!= AM_HAL_STATUS_-
SUCCESS )
{
    am_util_stdio_printf("Error enabling the SDIO power domain.\n);
}
if ( am_hal_pwrctrl_periph_disable(AM_HAL_PWRCTRL_PERIPH_SDIO)!= AM_HAL_STATUS_-
SUCCESS )
{
    am_util_stdio_printf("Error disabling the SDIO power domain.\n);
}
```

There is also a function to query if a peripheral is enabled:

```
am_hal_pwrctrl_periph_enabled(ePeripheral, &bEnabled);
```

3.13.4 Overall low-power initialization

The PWRCTRL HAL provides a single function call to configure the MCU into a standard low-power state. Examples can further reduce power if necessary, but this provides a good starting point for most applications.

```
am_hal_pwrctrl_low_power_init();
```

3.14 Real Time Clock (RTC)

The RTC is initialized by selecting and enabling the clock source. In the case of RTC the clocks can be either LFRC or External Oscillator (XT).

```
//
// Enable the LFRC for the RTC.
//
am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_LFRC_START, 0);

//
// Select LFRC for RTC clock source.
//
am_hal_rtc_osc_select(AM_HAL_RTC_OSC_LFRC);
```

The time can be set using the **am_hal_rtc_time_set(&hal_time)**. Notice the **hal_time** variable is a structure of **am_hal_rtc_time_t** type.

The **hal_time** structure variable can be set to the exact time from an external source (e.g., mobile phone time over Bluetooth Low Energy). In our examples, we set it to either an arbitrary date and time or to the date and time strings obtained from the compiler at compiler time.

```
   //
   // set the time and date using the date and time strings
   //
    hal_time.ui32Hour = 09;
    hal_time.ui32Minute = 00;
    hal_time.ui32Second = 00;
    hal_time.ui32Hundredths = 00;
    hal_time.ui32Weekday = 2;
    hal_time.ui32DayOfMonth = 26;
    hal_time.ui32Month = 4;
    hal_time.ui32Year = 22;
    hal_time.ui32Century = 21;
```

After setting the specific date and time in the structure, the **am_hal_rtc_time_set(&hal_time)** is called to set the time in the RTC.

Once the RTC has been set and started the **am_hal_rtc_time_get(&hal_time)** can be used to receive the value for the current time and date.

The RTC HAL also provides alarm functionality which can be set using the **am_hal_rtc_alarm_set(&g_sAlarmTime, AM_HAL_RTC_ALM_RPT_DAY)**. Here, the **g_sAlarmTime** variable is a structure of **am_hal_rtc_time_t** type and **AM_HAL_RTC_ALM_RPT_DAY** is defined as the desired alarm repeat interval.

3.15 Secure Digital Input Output (SDIO)

The SDIO HAL only supports the eMMC device and it consists of three layers. The eMMC card protocol **layer(am_hal_card.c)**, the generic card host abstraction **layer (am_hal_card_host.c)** and the hardware host controller **layer(am_hal_sdhc.c)**. The eMMC card protocol layer provides APIs for eMMC card operations (initialization, read, write, erase, etc). The generic card host abstraction layer is a glue layer to link protocol driver and hardware host controller (SDHC or SPI). Currently hardware host controller layer only supports SDHC host controller and it deals with hardware configuration, sending command, receiving response and data transfer.

SDIO initialization for eMMC card is as follows. The initialization includes host controller initialization, finding the card, card initialization and card configuration.

- Define a global card host pointer

```
am_hal_card_host_t *pSdhcCardHost = NULL;
```

- Initialize the card host. Currently only one SDHC card host is supported, so always use **AM_HAL_SDHC_CARD_HOST**, 'true' parameter. This forces the SDHC card host to reinitialize from scratch. Using 'false' will not force the SDHC card host to do reinitialization.

```
pSdhcCardHost = am_hal_get_card_host(AM_HAL_SDHC_CARD_HOST, true);
if (pSdhcCardHost == NULL)
{
am_util_stdio_printf("No such card host and stopped!!!\n");
while(1);
}
```

- After SDHC card host controller is initialized, define an eMMC card struct and check for card presence.

```
am_hal_card_t eMMCard;
While (am_hal_card_host_find_card (pSdhcCardHost, &eMMCard) !=
AM_HAL_STATUS_SUCCESS)
{
am_util_stdio_printf("No card is present now\n");
am_util_delay_ms(1000);
am_util_stdio_printf("Checking if card is available again\n");
}
```

- Initialize eMMC card. The power control policy is user defined. There are three policies to choose for different power consumption and throughput performance. The power control functions (**am_hal_card_pwrctrl_sleep** and **am_hal_card_pwrctrl_wake**) should be called if appropriate power policy is chosen. After initialization, the eMMC card is running with the default setting - 1-bit bus width and lowest clock speed

```
while (am_hal_card_init(&eMMCard, NULL, AM_HAL_CARD_PWR_CTRL_SDHC_OFF) !=
AM_HAL_STATUS_SUCCESS)
{
am_util_delay_ms(1000);
am_util_stdio_printf("card and host is not ready, try again\n");
}
```

- Configure the eMMC card. Configure the bus width, bus speed, bus voltage and eMMC bus mode according to the hardware design and eMMC chip.

```
while (am_hal_card_cfg_set(&eMMCard, AM_HAL_CARD_TYPE_EMMC,
AM_HAL_HOST_BUS_WIDTH_4, 48000000, AM_HAL_HOST_BUS_VOLTAGE_1_8,
AM_HAL_HOST_UHS_DDR50) != AM_HAL_STATUS_SUCCESS)
{
     am_util_delay_ms(1000);
     am_util_stdio_printf("setting DDR50 failed\n");
}
```

- To perform synchronous block read and write, the starting sector (block) and number of sectors (blocks) should be specified. The sector (block) size is 512 bytes and thus the read and write buffer size should be at least 512*number of sectors(blocks). The 32-bits return value includes the number of sectors(blocks) successfully transferred in upper 16-bits and transfer status in lower 16-bits. A synchronous block read & write will return from the function call when the entire transfer is complete.

```
ui32Status  =  am_hal_card_block_write_sync(&eMMCard, START_BLK, BLK_NUM,
(uint8_t *) pui8WrBuf);

am_util_stdio_printf("Synchronous Writing %d blocks is done, Xfer Status
%d\n", ui32Status >> 16, ui32Status & 0xffff);

ui32Status  =  am_hal_card_block_read_sync(&eMMCard, START_BLK, BLK_NUM,
(uint8_t *)pui8RdBuf);
am_util_stdio_printf("Synchronous Reading %d blocks is done, Xfer Status
%d\n", ui32Status >> 16, ui32Status & 0xffff);
```

- In addition, SDIO HAL supports asynchronous block read and write, the asynchronous transfers notify the application through the callback mechanism and interrupts. The SDIO interrupt service routine and a user callback function need to be defined.

```
void am_sdio_isr(void)
{
    uint32_t ui32IntStatus;
    am_hal_sdhc_intr_status_get(pSdhcCardHost->pHandle, &ui32IntStatus,
true);
    am_hal_sdhc_intr_status_clear(pSdhcCardHost->pHandle, ui32IntStatus);
    am_hal_sdhc_interrupt_service(pSdhcCardHost->pHandle, ui32IntStatus);
}

//register the callback
am_hal_card_register_evt_callback(&eMMCard, am_hal_card_event_test_cb);

//user callback
```

```
void am_hal_card_event_test_cb(am_hal_host_evt_t *pEvt)
{
    am_hal_card_host_t *pHost = (am_hal_card_host_t *)pEvt->pCtx;
    if (AM_HAL_EVT_XFER_COMPLETE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_READ)
    {
        bAsyncReadIsDone = true;
       am_util_debug_printf("Last Read Xfered block %d\n", pEvt->ui32BlkCnt);
    }
    if (AM_HAL_EVT_XFER_COMPLETE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_WRITE)
    {
        bAsyncWriteIsDone = true;
        am_util_debug_printf("Last Write Xfered block %d\n", pEvt-
>ui32BlkCnt);
    }
    if (AM_HAL_EVT_CARD_PRESENT == pEvt->eType)
    {
        am_util_stdio_printf("A card is inserted\n");
    }
}

//write blk_cnt asynchronously to eMMC
bAsyncWriteIsDone = false;
ui32Status = am_hal_card_block_write_async(&eMMCard, start_blk, blk_cnt,
(uint8_t *)ui8WrBuf);

//read blk_cnt asynchronously from eMMC
bAsyncReadIsDone = false;
ui32Status = am_hal_card_block_read_async(&eMMCard, start_blk, blk_cnt,
(uint8_t *)ui8RdBuf);
```

- Due to board layout and other factors, high frequency transfers may require a timing calibration to assure robust communication. A calibration function is available which allows the user to determine the proper timing settings for the SDIO interface. Note that the calibration function should come before the SDIO initialization process. After calibration, the timing setting is directly applied.

```
am_hal_card_emmc_calibrate(AM_HAL_HOST_UHS_DDR50, 48000000,
AM_HAL_HOST_BUS_WIDTH_8, (uint8_t *)write_buf, START_BLK, 2, ui8TxRxDelays);
```

  To set the timing settings directly, use the following function.

```
am_hal_card_host_set_txrx_delay(pSdhcCardHost, ui8TxRxDelays);
```

3.16 System Timer (STIMER)

The STIMER is a System Timer for Apollo devices and is a 32bit counter register which is reset on the actual power reset of the system (POR), but not on the SW initiated reset (SWPOR).

The STIMER HAL has four major functions:

- Configuration
- Capture

- Compare
- Interrupt Control

The following clocks can be used as input for the STIMER:

- NOCLK = 0x0 - No clock enabled.
- HFRC_6MHZ = 0x1 - 6MHz from the HFRC clock divider.
- HFRC_375KHZ = 0x2 - 375KHz from the HFRC clock divider.
- XTAL_32KHZ = 0x3 - 32768Hz from the crystal oscillator.
- XTAL_16KHZ = 0x4 - 16384Hz from the crystal oscillator.
- XTAL_1KHZ = 0x5 - 1024Hz from the crystal oscillator.
- LFRC_1KHZ = 0x6 - Approximately 1KHz from the LFRC oscillator (uncalibrated).
- CTIMER0 = 0x7 - Use CTIMER 0 for the clock source (allows prescaling from other system clocks).
- CTIMER1 = 0x8 - Use CTIMER 1 for the clock source (allows prescaling from other system clocks).

These configurations are implemented using the **am_hal_stimer_config(uint32_t ui32STimerConfig)** where the **ui32STIMERConfig** defines the configuration for the STIMER.

The **am_hal_stimer_counter_get()** function is used to get the current reading of the STIMER counter register. Note, that before reading the STIMER counter register the STIMER must be frozen then and only then the value can be read safely.

```
//
// Stop SysTick
//
am_hal_stimer_config(ui32Speed | AM_HAL_STIMER_CFG_FREEZE);
StimerVal = am_hal_stimer_counter_get();
```

The STIMER Capture Control Register controls each of the 4 capture registers. It selects their GPIO pin number for a trigger source, enables a capture operation and sets the input polarity for the capture.

NOTE: 8-bit writes can control individual capture registers atomically.

The **am_hal_stimer_capture_start/stop(uint32_t ui32CaptureNum, uint32_t ui32GPIONumber, bool bPolarity)** sets up the 4 capture register settings and capture **value. am_hal_stimer_capture_get(uint32_t ui32CaptureNum)** returns the capture value.

The STIMER Compare register is used to compare against the VALUE in the COUNTER register. If the match criterion in the configuration register is met, then a corresponding interrupt status bit is set. The match criterion is defined as COUNTER equal to COMPARE. To establish a desired value in this COMPARE register, write the number of ticks in the future to this register to indicate when to interrupt. The hardware does the addition to the COUNTER value in the STIMER clock domain so that the math is precise. Reading this register shows the COUNTER value at which this interrupt will occur.

The following function is used to set the delta value.

```
//
// Set a compare value for the future, and check to see what the absolute
// value of the compare register gets set to.
//
ui32Ret = am_hal_stimer_compare_delta_set(ui32CmprInstance, ui32Delta);
```

The following function is used to get the STIMER value once the interrupt is fired upon comparison with compare value.

```
ui32CompValue = am_hal_stimer_compare_get(ui32CmprInstance);
```

The STIMER NVRAM register contains a portion of the stored epoch offset associated with the time in the COUNTER register. This register is only reset by POI not by HRESETn. Its contents are intended to survive all reset level except POI and full power cycles.

It can be configured and read using the **am_hal_stimer_nvram_set/get(uint32_t ui32NvramNum, uint32_t ui32NvramVal)** respectively.

am_hal_stimer_int_enable/disable(uint32_t ui32Interrupt) enable/disable the following STIMER interrupts:

- `AM_HAL_STIMER_INT_COMPAREA`
- `AM_HAL_STIMER_INT_COMPAREB`
- `AM_HAL_STIMER_INT_COMPAREC`
- `AM_HAL_STIMER_INT_COMPARED`
- `AM_HAL_STIMER_INT_COMPAREE`
- `AM_HAL_STIMER_INT_COMPAREF`
- `AM_HAL_STIMER_INT_COMPAREG`
- `AM_HAL_STIMER_INT_COMPAREH`
- `AM_HAL_STIMER_INT_OVERFLOW`
- `AM_HAL_STIMER_INT_CAPTUREA`
- `AM_HAL_STIMER_INT_CAPTUREB`
- `AM_HAL_STIMER_INT_CAPTUREC`
- `AM_HAL_STIMER_INT_CAPTURED`

3.17 Timer (TIMER)

The Timer has 16 instances all of which can be configured to run with different clock sources.

The following clocks can be used as a source clock for a timer.

- `HFRC_DIV4 = 0x0 - Clock source is the HFRC / 4`
- `HFRC_DIV16 = 0x1 - Clock source is HFRC / 16`
- `HFRC_DIV64 = 0x2 - Clock source is HFRC / 64`
- `HFRC_DIV256 = 0x3 - Clock source is HFRC / 256`
- `HFRC_DIV1024 = 0x4 - Clock source is HFRC / 1024`
- `HFRC_DIV4K = 0x5 - Clock source is HFRC / 4096`
- `LFRC = 0x6 - Clock source is LFRC`

- `LFRC_DIV2 = 0x7 - Clock source is LFRC / 2`
- `LFRC_DIV32 = 0x8 - Clock source is LFRC / 32`
- `LFRC_DIV1K = 0x9 - Clock source is LFRC / 1024`
- `XT = 0xA - Clock source is the XT (uncalibrated).`
- `XT_DIV2 = 0xB - Clock source is XT / 2`
- `XT_DIV4 = 0xC - Clock source is XT / 4`
- `XT_DIV8 = 0xD - Clock source is XT / 8`
- `XT_DIV16 = 0xE - Clock source is XT / 16`
- `XT_DIV32 = 0xF - Clock source is XT / 32`
- `XT_DIV128 = 0x10 - Clock source is XT / 128`
- `RTC_100HZ = 0x11 - Clock source is 100 Hz from the current RTC oscillator.`

Along with these clocks, the OUT0 and OUT1 of timer can also be used as an input to other timer as well as GPIO's can be used to provide external clocks to the timers.

am_hal_timer_config(uint32_t ui32TimerNumber, am_hal_timer_config_t *psTimerConfig) can be used to configure not only the clock but also timer function, output inversion setting, trigger type and source, pattern limit and compare register values.

The following timer functions are available for configuration:

- **EDGE** - This Mode generates a single edge on OUT0/OUT1 when TIMER value hits CMP0/CMP1 respectively. OUT[0]=0, counter increments to CMP0, OUT[0]=1, counter stops. OUT[1] follows CMP1.
- **UPCOUNT** - This mode is run up counter generating a pulse on CMP. OUT[0]/OUT[1] is pulsed for one source clock period when TIMER matches CMP0/CMP1 respectively. Timer repeats for TMR_LMT iterations.
- **PWM** – `OUT0` and `OUT1` are waveforms, and not just one clock pulse. CMP1 dictates the low phase of the output and CMP0 dictates the period. OUT[1]=~OUT[0].

  SINGLEPATTERN = 0xC - Single pattern. OUT0=CMP0[TIMER], OUT1=CMP1[TIMER]. LMT field specifies length of pattern. When LMT GT 32 OUT0 and OUT1 is the same 64-bit pattern consisting of concatenated CMP1, CMP0. When LMT LT 32 OUT0 and OUT1 are independent. Both OUT0 and OUT1 can be inverted individually applications with POL0/POL1 = 0x1.
- **REPEATPATTERN** - Repeated pattern. Like SINGLEPATTERN mode, but pattern repeats after reaching LMT.

am_hal_timer_enable(ui32TimerNum) API is used to enable and start the timer once the setting has been configured into the timer.

am_hal_timer_disable(ui32TimerNum) API is used to disable and stop the timer.

am_hal_timer_read(ui32TimerNum) API is used to read the timer value. Though cation should be used before reading the timer by stopping the timer before it is read.

am_hal_timer_interrupt_enable/disable(ui32InterruptMask) is used to set the interrupts of the timer.

am_hal_timer_compare0/1_set(uint32_t ui32TimerNumber,uint32_t ui32-CompareValue) is used to set the compare registers. COMPARE1 is used to generate interrupts and output level shifts for a timer value between zero and COMPARE0. Check the description of your selected TIMER mode for a precise description of the function of COMPARE1. This change is done "on the fly" without disabling the timer for use with the DOWNCOUNT function.

3.18 UART

UART is initialized with **am_hal_uart_initialize(psConfig->uart, g_hUART)** where **psConfig->uart** is the UART number since there are 4 instances of the UART namely UART0, UART1, UART2 and UART3. **g_hUART** is a pointer which handles the current state if the UART in use.

The UART can be configured to different settings such as the baud rate, length of data, parity, stop bits, flow control, TX FIFO level and RX FIFO level. This can be achieved using the **am_hal_uart_configure(g_hUART, &sUARTConfig)**.

The HAL also provides a method to configure the buffer for UART communication which is **am_hal_uart_buffer_configure(void *pHandle, uint8_t *pui8TxBuffer, uint32_t ui32TxBufferSize, uint8_t *pui8RxBuffer, uint32_t ui32RxBufferSize)**. The buffer is implemented as a queue.

The UART has 11 different interrupt available which are as follows:

Table 3-1: UART Interrupts

#	Interrupt	Type	Description
10	OEIM	RW	This bit holds the overflow interrupt enable.
9	BEIM	RW	This bit holds the break error interrupt enable.
8	PEIM	RW	This bit holds the parity error interrupt enable.
7	FEIM	RW	This bit holds the framing error interrupt enable.
6	RTIM	RW	This bit holds the receive timeout interrupt enable.
5	TXIM	RW	This bit holds the transmit interrupt enable.
4	RXIM	RW	This bit holds the receive interrupt enable.
3	DSRMIM	RW	This bit holds the modem DSR interrupt enable.
2	DCDMIM	RW	This bit holds the modem DCD interrupt enable.
1	CTSMIM	RW	This bit holds the modem CTS interrupt enable.
0	TXCMPMIM	RW	This bit holds the modem TXCMP interrupt enable.

These interrupts can be enabled/disabled using the **am_hal_uart_interrupt_enable/disable(g_hUART, "interrupt mask")**.

The transmission or reception starts once the **am_hal_uart_transfer(g_hUART, &sTransaction)** where **sTransaction** is a structure of type **am_hal_uart_transfer_t** which describes the configuration of the data.

3.19 Universal Serial Bus (USB)

The SDK allows the user to develop USB device application on top of the TinyUSB. TinyUSB is an open source small footprint USB host stack. It is integrated with the SDK as an example implementation, but Ambiq does not guarantee its functionality. TinyUSB handles most of the high-level USB protocol and relies on the USB device controller of Apollo4 devices for data transactions on different endpoints.

The SDK allows the user to develop USB device application on top of the TinyUSB. TinyUSB is an opensource small footprint USB host stack. It is integrated with the SDK as an example implementation, but Ambiq does not guarantee its functionality.

The USB device controller is initialized in **dcd_init()** of TinyUSB porting **dcd_apollo4.c**, along with the USB power rails and the USB PHY. The users may need to change the enablement of external power rails according to your board settings. The default USB device event callback and the endpoint callbacks are registered here also.

At the application level, an USB example is usually structured into 3 files: **tusb_config.h** which contains the configuration, usb_descriptors.c which contains the descriptors and their related callbacks, and **<example name>.c** which contains the main logic and the implementation of callbacks

Configuration – tusb_config.h

It defines the USB class and the corresponding FIFOs' size etc. The below is the configuration for the CDC device used in the example **tinyusb_cdc**.

```
//------------ CLASS -------------//
#define CFG_TUD_CDC             1
#define CFG_TUD_MSC             0
#define CFG_TUD_HID             0
#define CFG_TUD_MIDI            0
#define CFG_TUD_VENDOR          0


// CDC FIFO size of TX and RX
#define CFG_TUD_CDC_RX_BUFSIZE   (TUD_OPT_HIGH_SPEED ? 512 : 64)
#define CFG_TUD_CDC_TX_BUFSIZE   (TUD_OPT_HIGH_SPEED ? 512 : 64)
```

Descriptors – usb_descriptors.c

The descriptors need to be provided by implementing the following 3 callback methods:

```
uint8_t const * tud_descriptor_device_cb(void)
uint8_t const * tud_descriptor_configuration_cb(uint8_t index)
uint16_t const* tud_descriptor_string_cb(uint8_t index, uint16_t langid)
```

The **tud_descriptor_device_cb** provides the device profile; The **tud_descriptor_-configuration_cb** provides the concatenated configuration, interface and end-point profiles;

The **tud_descriptor_string_cb** is responsible to provide the strings in UTF8.

All the complexity in the **usb_descriptors.c** is related to the definition of the descriptors.

The Program Logic – <example name>.c

The basic anatomy of the TinyUSB application logic is quite simple:

board_init() setup the board, you may refer to our default implementation in the example;

tusb_init() setup the USB functionality, you may our default implementation in TinyUSB porting **third_party\tinyusb\src\tusb.c**

And you need to call **tud_task()** – as often as possible to make TinyUSB stack work.

The remaining parts are application specific: you might want to send some messages via CDC and blink your leds, like the example **tinyusb_cdc** does.

```
int main(void)
{
  board_init();
  tusb_init();

  while (1)
  {
    tud_task(); // tinyusb device task
    led_blinking_task();
  }

  return 0;
}
```

In order to be able to react to some events, the following callbacks are provided by the framework:

```
void tud_mount_cb(void) // Invoked when device is mounted
void tud_umount_cb(void) // Invoked when device is unmounted
void tud_suspend_cb(bool remote_wakeup_en) // Invoked when usb bus is suspended
void tud_resume_cb(void) // Invoked when usb bus is resumed
```

In the example **tinyusb_cdc**, these callback methods are used to change the LED blinking speed.

3.20 Watchdog Timer (WDT)

The watchdog on Apollo devices runs on the LFRC clock which can be switched on using the **am_hal_clkgen_control(AM_HAL_CLKGEN_CONTROL_LFRC_START, 0)**.

The WDT can be configured using the **am_hal_wdt_config(AM_HAL_WDT_MCU, &g_sWatchdogConfig);** where **g_sWatchdogConfig** is the configuration structure for the WDT.

The **g_sWatchdogConfig** can configure the following memebers:

- **WDTEN** - Enable the WDT
- **INTEN** - Enable the WDT interrupt
- **RESEN** – Reset the WDT
- **RESVAL** - This bitfield is the compare value for counter bits 7:0 to generate a watchdog reset. This will cause a software reset.
- **INTVAL** - This bitfield is the compare value for counter bits 7:0 to generate a watchdog interrupt.

The next step is to enable the WDT interrupt utilizing **am_hal_wdt_interrupt_enable(AM_HAL_WDT_MCU, AM_HAL_WDT_INTERRUPT_MCU);** as well as setting the master interrupt in the NVIC.

The **am_hal_wdt_start(AM_HAL_WDT_MCU, false);** start the WDT as well as lock the WDT if required so that the setting for the WDT cannot be modified during execution of the program.

The AmbiqSuite SDK provides a GDB script under the tools directory which can be used to halt the watchdog during debug.

The script has the following methods:

am_attach [WDT_ON]

Attach to the Ambiq target and disable the watchdog. If **WDT_ON** is given as "1", don't disable the watchdog.

NOTE: If you call this function to attach and disable the watchdog, you should also call **am_detach** to end your GDB session. Otherwise, the watchdog will remain disabled until **am_detach** is called, or power is removed.

am_detach [WDT_ON]

Detach from the Ambiq target, immediately ending the GDB session. If **WDT_ON** is given as "1", also manually enable the watchdog timer.

Regardless of the **WDT_ON** argument, the Ambiq firmware will have full control of the watchdog timer starting from the next reset after **am_detach** is called.

A-SOCAP4-UGGA06EN v1.0
August 2021