USER'S GUIDE

Apollo3 Porting Guide

Ultra-Low Power Apollo SoC Family

A-SOCAP3-UGGA02EN v1.1

Legal Information and Disclaimers

AMBIQ MICRO INTENDS FOR THE CONTENT CONTAINED IN THE DOCUMENT TO BE ACCURATE AND RELIABLE. THIS CONTENT MAY, HOWEVER, CONTAIN TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS OR OTHER MISTAKES. AMBIQ MICRO MAY MAKE CORRECTIONS OR OTHER CHANGES TO THIS CONTENT AT ANY TIME. AMBIQ MICRO AND ITS SUPPLIERS RESERVE THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS AND OTHER CHANGES TO ITS PRODUCTS, PROGRAMS AND SERVICES AT ANY TIME OR TO DISCONTINUE ANY PRODUCTS, PROGRAMS, OR SERVICES WITHOUT NOTICE.

THE CONTENT IN THIS DOCUMENT IS PROVIDED "AS IS". AMBIQ MICRO AND ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THIS CONTENT FOR ANY PURPOSE AND DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS CONTENT, INCLUDING BUT NOT LIMITED TO, ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHT.

AMBIQ MICRO DOES NOT WARRANT OR REPRESENT THAT ANY LICENSE, EITHER EXPRESS OR IMPLIED, IS GRANTED UNDER ANY PATENT RIGHT, COPYRIGHT, MASK WORK RIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO COVERING OR RELATING TO THIS CONTENT OR ANY COMBINATION, MACHINE, OR PROCESS TO WHICH THIS CONTENT RELATE OR WITH WHICH THIS CONTENT MAY BE USED.

USE OF THE INFORMATION IN THIS DOCUMENT MAY REQUIRE A LICENSE FROM A THIRD PARTY UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF THAT THIRD PARTY, OR A LICENSE FROM AMBIQ MICRO UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF AMBIQ MICRO.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE SYSTEM AND SOFTWARE IMPLEMENTERS TO USE AMBIQ MICRO PRODUCTS. THERE ARE NO EXPRESS OR IMPLIED COPYRIGHT LICENSES GRANTED HEREUNDER TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS OR INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CHANGES WITHOUT FURTHER NOTICE TO ANY PRODUCTS HEREIN. AMBIQ MICRO MAKES NO WARRANTY, REPRESENTATION OR GUARANTEE REGARDING THE SUITABILITY OF ITS PRODUCTS FOR ANY PARTICULAR PURPOSE, NOR DOES AMBIQ MICRO ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR CIRCUIT, AND SPECIFICALLY DISCLAIMS ANY AND ALL LIABILITY, INCLUDING WITHOUT LIMITATION CONSEQUENTIAL OR INCIDENTAL DAMAGES. "TYPICAL" PARAMETERS WHICH MAY BE PROVIDED IN AMBIQ MICRO DATA SHEETS AND/OR SPECIFICATIONS CAN AND DO VARY IN DIFFERENT APPLICATIONS AND ACTUAL PERFORMANCE MAY VARY OVER TIME. ALL OPERATING PARAMETERS, INCLUDING "TYPICALS" MUST BE VALIDATED FOR EACH CUSTOMER APPLICATION BY CUSTOMER'S TECHNICAL EXPERTS. AMBIQ MICRO DOES NOT CONVEY ANY LICENSE UNDER NEITHER ITS PATENT RIGHTS NOR THE RIGHTS OF OTHERS. AMBIQ MICRO PRODUCTS ARE NOT DESIGNED, INTENDED, OR AUTHORIZED FOR USE AS COMPONENTS IN SYSTEMS INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE AMBIQ MICRO PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR. SHOULD BUYER PURCHASE OR USE AMBIQ MICRO PRODUCTS FOR ANY SUCH UNINTENDED OR UNAUTHORIZED APPLICATION, BUYER SHALL INDEMNIFY AND HOLD AMBIQ MICRO AND ITS OFFICERS, EMPLOYEES, SUBSIDIARIES, AFFILIATES, AND DISTRIBUTORS HARMLESS AGAINST ALL CLAIMS, COSTS, DAMAGES, AND EXPENSES, AND REASONABLE ATTORNEY FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PERSONAL INJURY OR DEATH ASSOCIATED WITH SUCH UNINTENDED OR UNAUTHORIZED USE, EVEN IF SUCH CLAIM ALLEGES THAT AMBIQ MICRO WAS NEGLIGENT REGARDING THE DESIGN OR MANUFACTURE OF THE PART.

Revision History

Revision	Date	Description
0.1	September 1, 2018	Initial release
0.2	September 10, 2018	Updates
0.3	September 14, 2018	Fast GPIO
1.0	March 6, 2022	Template update
1.1	January 3, 2023	Updated the document part number

Reference Documents

Document ID	Description

Table of Contents

List of Tables

SECTION

1

Overview

This document is a guide to porting applications from the Apollo/Apollo2 SDK to the Apollo3 SDK. The Apollo3 SDK is a step towards incorporating more industry standard features into the AmbiqSuite. Specifically, the Apollo3 SDK supports:

- CMSIS standard register definitions, interrupt vectors, and intrinsics
- Uniform device driver model for communications APIs
- Consistent error checking and error code returns
- Device instances associated with device handle

SECTION

2

CMSIS Register Definitions

The Apollo3 SDK supports the Cortex Microcontroller Software Interface Standard or CMSIS. Among other things, CMSIS defines a standard syntax for exposing SoC register definitions to software, interrupt vector naming and intrinsic function exposure. The Apollo SDK will deprecate the AM_REG (Ambiq Micro Register) macros at the production release of the SDK. Any code from Apollo/Apollo2 implementations that directly use registers will need to port to the CMSIS standard.

SECTION

3

Uniform Driver API

The majority of industry Cortex-M SDK implementations provide some level of commonality between similar devices (I^2C, I^2S, SPI, CAN, and UART). The Apollo/Apollo2 SDK organically grew the specific HAL device APIs based on the underlying features of the hardware and the need for efficient operation to demonstrate low-power. While this is necessary, it complicates the support model for many new customers as they need to learn the approach to each of these device APIs from scratch. The Uniform Device API is designed to establish a uniform set of APIs and requirement on the ADC, Bluetooth Low Energy, IOM, IOS, MSPI, PDM and UART devices in the Apollo3 SoC. These uniform APIs include the following generic functions:

- **initialize** – creates an instance of the given device interface and returns a handle to the instance for use with all other APIs.
- **power_control** – provides a consistent interface to power-up and sleep the device (typically a device cannot be configured without first calling power_control).
- **configure** – configures the entire device or a subset of the device resources (there can be multiple configure functions depending how the device provides resources).
- **enable** – enables a configured device to begin operation
- **control** – provides the ability to configure less used parameters or modes of the device
- **interrupt_enable** – enable one or more interrupts from the device.
- **interrupt_disable** - disable one or more interrupts from the device.
- **interrupt_status_get** – retrieve the current interrupts registered with the device.
- **interrupt_clear** – clear one or more interrupts from the device.
- **interrupt_service** – service an interrupt from the device.
- **disable** – disables a configured device to stop operation
- **deinitialize** – resets and returns a device instance

Not all devices require the full set as outlined above, but they do follow the same general pattern for initialization and shutdown. The differences are primarily in the calls made during operation of the device.

SECTION

4

HAL Status Return Values

There is a generic set of return values that is used by all HAL routines included in am_hal_status.h. In some cases these values are extended for driver specific error codes. These values are:

```
//
// Global Status Returns
//
typedef enum
{
   AM_HAL_STATUS_SUCCESS,
   AM_HAL_STATUS_FAIL,
   AM_HAL_STATUS_INVALID_HANDLE,
   AM_HAL_STATUS_IN_USE,
   AM_HAL_STATUS_TIMEOUT,
   AM_HAL_STATUS_OUT_OF_RANGE,
   AM_HAL_STATUS_INVALID_ARG,
   AM_HAL_STATUS_INVALID_OPERATION,
   AM_HAL_STATUS_MEM_ERR,
   AM_HAL_STATUS_HW_ERR,
   AM_HAL_STATUS_MODULE_SPECIFIC_START = 0x08000000,
} am_hal_status_e;
```

SECTION

5

HAL Optional API Validation

Several of the HAL drivers include optional code to validate parameters passed to the functions. While this code provides for additional error checking it also may cost efficiency. This code may be disabled by defining **AM_HAL_DISABLE_API_VALIDATION** in the compiled project.

SECTION

6

ADC HAL

6.1 Configuration

All of the selectable configuration structures have been converted into enums instead of #defines for value definitions to aid in compile-time type safety. For example:

```
#define AM_HAL_ADC_LPMODE_0               AM_REG_ADC_CFG_LPMODE_MODE0
#define AM_HAL_ADC_LPMODE_1               AM_REG_ADC_CFG_LPMODE_MODE1

typedef enum
{
  AM_HAL_ADC_LPMODE0, // Low Latency Clock Mode. When set, HFRC and the adc_clk
                      // will remain on while in functioning in LPMODE0.
  AM_HAL_ADC_LPMODE1  // Powers down all circuity and clocks associated with the
                      // ADC until the next trigger event. Between scans, the reference
                      // buffer requires up to 50us of delay from a scan trigger event
                      // before the conversion will commence while operating in this mode.
} am_hal_adc_lpmode_e;
```

Configuration of the ADC device can be mapped as shown in Table 6-1.

Table 6-1: ADC HAL Configuration Functional Map

Apollo/Apollo2 Function	Apollo3 Function
N/A	am_hal_adc_initialize
N/A	am_hal_adc_power_control
am_hal_adc_config	am_hal_adc_configure
am_hal_adc_window_set	am_hal_adc_control(AM_HAL_ADC_REQ_WINDOW_CONFIG)
am_hal_adc_slot_config	am_hal_adc_configure_slot
N/A	am_hal_adc_configure_dma
N/A	am_hal_adc_deinitialize

6.2 Operation

Operation of the ADC device can be mapped as shown in Table 6-2.

Table 6-2: ADC HAL Operation Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_adc_enable	am_hal_adc_enable
am_hal_adc_disable	am_hal_adc_disable
N/A	am_hal_adc_status_get
am_hal_adc_trigger	am_hal_adc_sw_trigger

6.3 Interrupts

Interrupt control of the ADC device can be mapped as shown in Table 6-3.

Table 6-3: ADC HAL Interrupt Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_adc_int_enable	am_hal_adc_interrupt_enable
am_hal_adc_int_disable	am_hal_adc_interrupt_disable
am_hal_adc_int_status_get	am_hal_adc_interrupt_status
am_hal_adc_int_clear	am_hal_adc_interrupt_clear

6.4 Data Movement

Apollo3 implements DMA to support the ADC device. With DMA the samples are transferred directly to SRAM and an interrupt is generated when the total number of samples is collected.

Raw data movement of the ADC device can be mapped as shown in Table 6-4.

Table 6-4: ADC HAL Data Movement Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_adc_fifo_peek	am_hal_adc_samples_read
am_hal_adc_fifo_pop	am_hal_adc_samples_read

SECTION

7

CTIMER HAL

7.1 Global Enable

There is a new function that allows synchronized start of all of the CTIMERs. This overrides the individual enables of the CTIMERs and is designed to be used with the new stepper motor/pattern generation features of the CTIMER. The default is for all the timers to be enabled.

7.2 Auxiliary Configuration

There are two additional functions that allow setting the new AUX registers. These are mostly designed to be used with the new stepper motor/pattern generation features of the CTIMER.

Table 7-1: CTIMER HAL Auxiliary Configuration Functional Map

Apollo3 Function
am_hal_ctimer_aux_read
am_hal_ctimer_aux_compare_set
am_hal_ctimer_aux_period_set

7.3 Output Control

There is a new function which provides for coordinate routing of the new CTIMER output signal scheme to I/O pads. The function enforces the restrictions on the pad connectivity. It is recommended that the customer study the tables in the function implementation and the datasheet to get a sense for these restrictions.

Table 7-2: CTIMER HAL Output Control Functional Map

Apollo3 Function
am_hal_ctimer_output_config

SECTION

8

GPIO HAL

The GPIO HAL implementation for Apollo3 has changed significantly from previous Apollo and Apollo2 SDKs. The hardware design for Apollo3 GPIO remains very similar to the previous products with a few new features added. The new GPIO features, however, do add some complexity that the new GPIO HAL makes more manageable for the user.

The previous software implementation was heavily dependent on macros for configuration and usage, which caused confusion to end users, especially with pin configuration, and contributed to code size due to the inline coding. The new implementation abstracts most of the configuration into the HAL with the caller supplying a single word containing all pin configuration parameters in a single 32-bit word defined by a standard C bitfield structure.

To further simplify pin definition, a tool is provided in order to easily define pin configurations. The pins are described in an ordinary text file and run through the script to produce compilable C code. See *Section 8.5 Create the bsp_pins.src File on page 22* for details.

8.1 Configuration

Table 8-1: GPIO HAL Configuration Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_gpio_pin_config	am_hal_gpio_pinconfig

This function is called when configuring a given pad for its ultimate function. The specified parameters (bfGpioCfg) are checked for compatibility with the specified pin. Any configuration or parameter errors result in an error return.

The prototype is am_hal_gpio_pinconfig(uint32_t ui32Pin, am_hal_gpio_pincfg_t bfGpioCfg).

The ui32Pin parameter is simply the pin number to be configured.

am_hal_gpio_pincfg_t is a bitfield structure containing the following members:

- **uFuncSel** - This is a value from 0-7 which will usually come from am_hal_pin.h.
- **ePullup** - Many pads can supply a pullup resistor. For those that do, this member defines the value of that pullup. It is one of the following enumerations:

```
//
// Pullup configuration: am_hal_gpio_pincfg_t.ePullup enums
//
typedef enum
{
    //
    // Define pullup enums.
    // The 1.5K - 24K pullup values are valid for select I2C enabled pads.
    // The "weak" value is used for almost every other pad except pin 20.
    //
    AM_HAL_GPIO_PIN_PULLUP_NONE = 0x00,
    AM_HAL_GPIO_PIN_PULLUP_WEAK,
    AM_HAL_GPIO_PIN_PULLUP_1_5K,
    AM_HAL_GPIO_PIN_PULLUP_6K,
    AM_HAL_GPIO_PIN_PULLUP_12K,
    AM_HAL_GPIO_PIN_PULLUP_24K,
    AM_HAL_GPIO_PIN_PULLDOWN
} am_hal_gpio_pullup_e;
```

- **eGPOutcfg** - This member is generally used when defining a pad as a GPIO output and defines the output type. It is one of the following enumerations:

```
//
// OUTCFG pad configuration: am_hal_gpio_pincfg_t.eGPOutcfg enums
// Applies only to GPIO configured pins.
// Ultimately maps to GPIOCFG.OUTCFG, bits [2:1].
//
typedef enum
{
    AM_HAL_GPIO_PIN_OUTCFG_DISABLE = 0x0,
    AM_HAL_GPIO_PIN_OUTCFG_PUSHPULL = 0x1,
    AM_HAL_GPIO_PIN_OUTCFG_OPENDRAIN = 0x2,
    AM_HAL_GPIO_PIN_OUTCFG_TRISTATE = 0x3
} am_hal_gpio_outcfg_e;
```

- **eDriveStrength** - For output configurations, many pads can be configured with various drive strengths. For those that do, this member defines that and will be one of the following enumerations:

```
//
// Pad Drive Strength configuration: am_hal_gpio_pincfg_t.eDriveStrength
enums
//
typedef enum
{
    //
    // DRIVESTRENGTH is a 2-bit field.
    // bit0 maps to bit2 of a PADREG field.
    // bit1 maps to bit0 of an ALTPADCFG field.
    //
    AM_HAL_GPIO_PIN_DRIVESTRENGTH_2MA = 0x0,
    AM_HAL_GPIO_PIN_DRIVESTRENGTH_4MA = 0x1,
    AM_HAL_GPIO_PIN_DRIVESTRENGTH_8MA = 0x2,
```

```
    AM_HAL_GPIO_PIN_DRIVESTRENGTH_12MA = 0x3
} am_hal_gpio_drivestrength_e;
```

- **eGPInput** - This member is generally used when defining a pad as a GPIO input and defines the input type. It is one of the following enumerations:

```
//
// GPIO input configuration: am_hal_gpio_pincfg_t.eGPInput enums
// Applies only to GPIO configured pins!
// Ultimately maps to PADREG.INPEN, bit1.
//
typedef enum
{
    AM_HAL_GPIO_PIN_INPUT_AUTO   = 0x0,
    AM_HAL_GPIO_PIN_INPUT_NONE   = 0x0,
    AM_HAL_GPIO_PIN_INPUT_ENABLE = 0x1
} am_hal_gpio_input_e;
```

- **eGPRdZero** - This member is generally used when defining a pad as a GPIO input and defines whether the pin value can be read or if it always reads as zero. It is one of the following enumerations:

```
//
// am_hal_gpio_pincfg_t.eGPRdZero
// For GPIO configurations (funcsel=3), the pin value can be read or 0 can be
// forced as the read value.
//
typedef enum
{
    AM_HAL_GPIO_PIN_RDZERO_READPIN   = 0x0,
    AM_HAL_GPIO_PIN_RDZERO_ZERO      = 0x1
} am_hal_gpio_readen_e;
```

- **eIntDir** - This member is used when interrupts are to be enabled for a pad. It is one of the following enumerations:

```
//
// GPIO interrupt direction configuration: am_hal_gpio_pincfg_t.eIntDir enums
// Note: Setting INTDIR_NONE has the side-effect of disabling being able to
// read a pin - the pin will always read back as 0.
//
typedef enum
{
    // Bit1 of these values maps to GPIOCFG.INCFG (b0).
    // Bit0 of these values maps to GPIOCFG.INTD (b3).
    AM_HAL_GPIO_PIN_INTDIR_LO2HI   = 0x0,
    AM_HAL_GPIO_PIN_INTDIR_HI2LO   = 0x1,
    AM_HAL_GPIO_PIN_INTDIR_NONE    = 0x2,
    AM_HAL_GPIO_PIN_INTDIR_BOTH    = 0x3
} am_hal_gpio_intdir_e;
```

- **ePowerSw** - A select number of pins can be configured to source or sink current (see datasheet for which pins support these functions). For pins that support it, it is one of the following enumerations:

```
//
// Power Switch configuration: am_hal_gpio_pincfg_t.ePowerSw enums
//
typedef enum
{
```

```
    AM_HAL_GPIO_PIN_POWERSW_NONE,
    AM_HAL_GPIO_PIN_POWERSW_VDD,
    AM_HAL_GPIO_PIN_POWERSW_VSS,
    AM_HAL_GPIO_PIN_POWERSW_INVALID,
} am_hal_gpio_powersw_e;
```

- **uIOMnum** - This member is used when a pad is defined to be a chip enable and designates the IO Master number (0-5) or MSPI (6) that the CE is to be used for. Most pads can be configured as a chip enable with each pad supporting 4 combinations of IOM/MSPI and channel numbers. See the datasheet for a table of these combinations. This member is always a value of 0-5 or 6.
- **uNCE** - This member is used when a pad is defined to be a chip enable and is used in conjunction with uIOMnum to define the CE number for a particular SPI device. It is always a value of 0-3.
- **eCEpol** - This member is used when a pad is defined to be a chip enable and specifies the polarity of the CE enable. It is one of the following enumerations:

```
//
// nCE polarity configuration: am_hal_gpio_pincfg_t.eCEpol enums
//
typedef enum
{
    AM_HAL_GPIO_PIN_CEPOL_ACTIVELOW   = 0x0,
    AM_HAL_GPIO_PIN_CEPOL_ACTIVEHIGH  = 0x1
} am_hal_gpio_cepol_e;
```

8.2 Operation

Table 8-2: GPIO HAL Operation Functional Map

Apollo/Apollo2 Functions	Apollo3 Function
am_hal_gpio_input_bit_read am_hal_gpio_output_bit_read am_hal_gpio_enable_bit_get	am_hal_gpio_state_read

The new Apollo3 **am_hal_gpio_state_read** function is used for reading GPIO values.

The prototype is am_hal_gpio_state_read(uint32_t ui32Pin, am_hal_gpio_read_type_e eReadType, uint32_t *pui32ReadState).

ui32Pin is the pin number to be read.

eReadType is one of the following enumerations:

```
typedef enum
{
    AM_HAL_GPIO_INPUT_READ,
    AM_HAL_GPIO_OUTPUT_READ,
    AM_HAL_GPIO_ENABLE_READ
} am_hal_gpio_read_type_e;
```

pui32ReadState is a pointer to the variable to receive the read value of the pin.

Table 8-3: GPIO HAL Functional Map

Apollo/Apollo2 Functions	Apollo3 Function
am_hal_gpio_out_bit_set am_hal_gpio_out_bit_clear am_hal_gpio_out_bit_toggle	am_hal_gpio_state_write

The new Apollo3 **am_hal_gpio_state_write** function is used for writing GPIO values.

The prototype is am_hal_gpio_state_write(uint32_t ui32Pin, am_hal_gpio_write_type_e eWriteType).

ui32Pin is the pin number to be read.

eWriteType is one of the following enumerations:

```
typedef enum
{
    AM_HAL_GPIO_OUTPUT_CLEAR,
    AM_HAL_GPIO_OUTPUT_SET,
    AM_HAL_GPIO_OUTPUT_TOGGLE,
    AM_HAL_GPIO_OUTPUT_TRISTATE_DISABLE,
    AM_HAL_GPIO_OUTPUT_TRISTATE_ENABLE,
    AM_HAL_GPIO_OUTPUT_TRISTATE_TOGGLE
} am_hal_gpio_write_type_e;
```

8.3 Interrupt Functions

As with other peripherals, pins configured as GPIOs can be configured to provide interrupts. The HAL provides several functions to support this functionality.

Table 8-4: GPIO HAL Interrupt Functional Map

Apollo/Apollo2 Functions	Apollo3 Function
am_hal_gpio_int_enable	am_hal_gpio_interrupt_enable
am_hal_gpio_int_enable_get	
am_hal_gpio_int_disable	am_hal_gpio_interrupt_disable
am_hal_gpio_int_clear	am_hal_gpio_interrupt_clear
am_hal_gpio_int_set	
am_hal_gpio_int_status_get	am_hal_gpio_interrupt_status_get
am_hal_gpio_int_service	am_hal_gpio_interrupt_service
am_hal_gpio_int_register	am_hal_gpio_interrupt_register
am_hal_gpio_int_polarity_bit_get	

The **am_hal_gpio_interupt_enable** function enables the given interrupt(s). Only bits 0-49 are valid in the mask.

The **am_hal_gpio_interrupt_disable** function disables the given interrupt(s). Only bits 0-49 are valid in the mask.

The **am_hal_gpio_interrupt_clear** function clears the given interrupt(s). Only bits 0-49 are valid in the mask. This function is often used in conjunction with am_hal_gpio_interrupt_status_get(), with the returned IntStatus used as the input to this function.

The **am_hal_gpio_interrupt_status_get** function returns the current interrupt status. It can return the status of every interrupt (bEnabledOnly=false) or the status of only those that are enabled (bEnabledOnly=true). The 64bit variable pointed to be pui64IntStatus contains the return status.

The **am_hal_gpio_interrupt_service** function is an overall service routine for GPIO interrupts. It is called by am_gpio_isr(), which also calls am_hal_gpio_interrupt_status_get() to use as an input parameter to this function. The general usage is that the application calls am_hal_gpio_interrupt_register() to register a callback routine that this routine will call when the registered interrupt occurs. The application also supplies the main handler, am_gpio_isr().

The **am_hal_gpio_interrupt_register** function is call by the application for registering specific handlers to specific GPIO interrupts.

8.4 GPIO Read and Write Macros

While the primary read and write functions will suffice for virtually all applications, there may be situations where minimal response time is required. To support these situations a set of macros are provided which provide minimal inline code for accessing GPIOs.

Advantages to usage of these macros include faster GPIO read or write access times, no function call overhead, and simple read return values.

Drawbacks to usage of these macros include no error checking, larger resultant code size, no guaranteed atomicity, and risk to general safety.

The "_read" macros are counterparts to the enumerations used for the am_hal_gpio_state_read() function.

Likewise, the "_set, _clear, _toggle) macros are counterparts to the enumerations used for the am_hal_gpio_state_write() function.

Table 8-5: GPIO Read and Write Macros

Apollo3 GPIO Macros
am_hal_gpio_input_read(n)
am_hal_gpio_output_read(n)
am_hal_gpio_enable_read(n)

Table 8-5: GPIO Read and Write Macros *(Continued)*

Apollo3 GPIO Macros
am_hal_gpio_output_clear(n)
am_hal_gpio_output_set(n)
am_hal_gpio_output_toggle(n)
am_hal_gpio_output_tristate_enable(n)
am_hal_gpio_output_tristate_disable(n)
am_hal_gpio_output_tristate_toggle(n)

8.5 Create the bsp_pins.src File

The file bsp_pins.src is a simple text file containing names, keywords, and values that describe each pin. The text file is subsequently provided as input to a Python script that generates two files: am_bsp_pins.c and am_bsp_pins.h. These two C files contain each of the pins bitfield structures that are passed along to am_hal_gpio_pinconfig().

8.5.1 Creating a .src File

Note - the .src file should contain no tab characters (only spaces).

Also, indentation is important. A tab indentation of 4 spaces is recommended.

Each pin entry takes the form:

```
pin
    name = UART_TX
    desc = This pin is the COM_UART transmit pin.
    pinnum = 35
    func_sel = AM_HAL_PIN_35_UART1TX
    drvstrength = 2
```

While there are about a dozen keywords (parameters) available, only the parameters required to define a pin need be included in any particular definition when defined globally. If defined in a local variable (stack), all unused fields must be specifically set to 0.

The keywords used in the file are:

Table 8-6: Keyword Description

Keyword	Description
name	The name to be used for the pin. This name will be used as a base for generating defines. Each pin name must be unique.
desc	Optional: A description, if provided, will appear in the generated header file.
funcsel	A value 0-7, or the equivalent AM_HAL_PIN_nn_xxxx macro from am_hal_pin.h. Note that the AM_HAL_PIN_nn_xxxx nomenclature is preferred.

Table 8-6: Keyword Description *(Continued)*

Keyword	Description
pinnum	The pin number for the pin being defined (0-49).
drvstrength	One of: 2, 4, 8, or 12. If not provided, 2 is default.
GPOutCfg	Typically used if the pin is being defined as GPIO (funcsel=3). One of: disable, pushpull, opendrain, tristate. * Also acceptable is a value 0-3, or a define.
GPinput	Only used if the pin is being defined as GPIO (funcsel=3). One of: true, false.
GPRdZero	One of readpin, zero (or true or false). intdir One of: none, lo2hi, hi2lo, either. Note - does not enable any interrupt. Only configures the direction for when it is enabled.
pullup	One of: none, 1_5K, 6K, 12K, 24K. Also acceptable is a define (e.g. AM_HAL_GPIO_PIN_PULLUP_1_5K).
PowerSw	One of: VDD or VSS. Also acceptable is a define (e.g. AM_HAL_GPIO_PIN_POWERSW_VDD).

The following 3 parameters only apply when the pin is being defined as a chip enable, e.g., a CE for a SPI or MSPI device.

8.5.2 Build New Pin Files from the .src

Each bsp directory contains a Makefile that can be used to completely rebuild the BSP by simply typing "make" on the command line. A rebuild might be required, for instance, if the .src file is updated or if a BSP C function is modified. The first step of the build process is the creation of the am_bsp_pins.c and am_bsp_pins.h files using the .src file as input. Once those two files have been created, the Makefile then builds the BSP itself.

Alternatively, the am_bsp_pins.c and am_bsp_pins.h can be manually created by using the script found at tools/bsp_generator/pinconfig.py. The script must be run twice, once to create the .c file and again to create the .h file. The basic command line is:

```
pinconfig.py bsp_pins.src C >am_bsp_pins.c
pinconfig.py bsp_pins.src H >am_bsp_pins.h
```

8.6 Fast GPIO

The Apollo3 SoC introduced an alternative method of setting and clearing GPIOs, termed Fast GPIO. The set and clear registers for Fast GPIO operation are architecturally situated near the SoC core (in the APBDMA named block) such that GPIO

accesses can be handled with minimal latency. The Apollo3 HAL supports these Apollo3 specific functions.

One of the intended usages of Fast GPIO is in "bit-banging" operations for up to 8 bits in parallel, with each pin controlled with a single bit in the SETCLEAR register. This set/clear methodology imposes a limitation that only certain pins can be controlled with each bit. That control can be seen in the following matrix (as well as a similar matrix in am_hal_gpio.h) that relates the control bit to the pins that can be controlled by that bit.

Table 8-7: Fast GPIO

BIT	Pin Controlled by Bit						
0	0	8	16	24	32	40	48
1	1	9	17	25	33	41	49
2	2	10	18	26	34	42	
3	3	11	19	27	35	43	
4	4	12	20	28	36	44	
5	5	13	21	29	37	45	
6	6	14	22	30	38	46	
7	7	15	23	31	39	47	

Fast GPIO pin configuration is similar to normal pin configuration, but a new function is provided to facilitate it. Further it is recommended that prior to configuring a pin that the state be initialized using am_hal_gpio_fastgpio_disable() and am_hal_gpio_fastgpio_clr()/set().

The prototype of the Fast GPIO pin configuration function is:

am_hal_gpio_fast_pinconfig(uint64_t ui64PinMask, am_hal_gpio_pincfg_t bfGpioCfg, uint32_t ui32Masks[]).

Where ui64PinMask is a mask of the pins to be configured.

For the most efficient access, the Fast GPIO implementation is supported by various macros instead of functions.

- **am_hal_gpio_fastgpio_enable(n)** – Typically used after pin configuration to enable fast gpio for the specified pin.
- **am_hal_gpio_fastgpio_disable(n)** – Disable fast gpio on the specified pin.
- **am_hal_gpio_fastgpio_set(n)** – Set the given pin high.
- **am_hal_gpio_fastgpio_clr(n)** – Clear the value on the given pin.
- **am_hal_gpio_fastgpio_setmsk(n)** – Set the given pins high.
- **am_hal_gpio_fastgpio_clrmsk(n)** – Clear the values on the given pins.

- **am_hal_gpio_fastgpio_wrval(val)** – Write a value to all of the Fast GPIO configured pins.

Finally, note that each specified pin must be on a unique row. Even though this restriction is not strictly enforced by am_hal_gpio_fast_pinconfig(),

For example, the following call will configure 8 pins, pins 48, 41,34, 27, 20, 13, 6, 15, for output of fast gpio.

am_hal_gpio_fast_pinconfig((uint64_t)0x000102040810A040, g_AM_HAL_GPIO_OUTPUT, 0);

Each pin would then need am_hal_gpio_fastgpio_enable(), after which the set and clr macros could be used.

Note in this example that pin 48 would be controlled by bit0, 41 by bit1, 34 by bit2, 27 by bit3, 20 by bit4, 13 by bit5, 6 by bit6, and 15 by bit 7.

SECTION

9

IOM HAL

The Apollo3 IOM HAL interface has been greatly simplified from Apollo/Apollo2. Support for a wide variety of blocking, queue, and nonblocking read/write operations has been reduced to just three transfer functions.

9.1 Configuration

All of the selectable configuration structures have been converted into enums instead of #defines for value definitions to aid in compile-time type safety. Configuration of the IOM device can be mapped as follows:

Table 9-1: IOM HAL Configuration Functional Map

Apollo/Apollo2 Function	Apollo3 Function
N/A	am_hal_iom_initialize
N/A	am_hal_iom_power_ctrl
am_hal_iom_config	am_hal_iom_configure
N/A	am_hal_iom_control
N/A	am_hal_iom_deinitialize

9.2 Operation

Operation of the IOM device can be mapped as follows:

Table 9-2: IOM HAL Operation Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_iom_enable	am_hal_iom_enable
am_hal_iom_disable	am_hal_iom_disable

Table 9-2: IOM HAL Operation Functional Map *(Continued)*

Apollo/Apollo2 Function	Apollo3 Function
am_hal_iom_status_get	am_hal_iom_status_get
am_hal_iom_error_status_get	N/A

9.3 Interrupts

Interrupts control of the IOM device can be mapped as follows:

Table 9-3: IOM HAL Interrupts Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_iom_int_enable	am_hal_iom_interrupt_enable
am_hal_iom_int_enable_get	N/A
am_hal_iom_int_disable	am_hal_iom_interrupt_disable
am_hal_iom_int_status_get	am_hal_iom_interrupt_status_get
am_hal_iom_int_set	N/A
am_hal_iom_int_clear	am_hal_iom_interrupt_clear
am_hal_iom_int_service	am_hal_iom_interrupt_service

9.4 Data Movement

Apollo3 implements DMA to support the IOM devices. With DMA the samples are transferred directly to SRAM and an interrupt is generated when the total number of samples is collected. In addition, Apollo3 supports a Command Queue for each IOM device. The Command Queue is used inside the non-blocking transfer function to provide queued request for DMA transfer.

The data movement of the IOM device can be mapped as follows:

Table 9-4: IOM HAL Data Movement Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_iom_spi_write_nq am_hal_iom_spi_read_nq am_hal_iom_i2c_write_nq am_hal_iom_i2c_read_nq	am_hal_iom_blocking_transfer
am_hal_iom_spi_write_nb am_hal_iom_spi_read_nb am_hal_iom_queue_spi_write am_hal_iom_queue_spi_read am_hal_iom_i2c_write_nb am_hal_iom_i2c_read_nb	am_hal_iom_nonblocking_transfer
am_hal_iom_spi_fullduplex_nq	N/A

The Apollo3 IOM HAL data movement operations have been greatly simplified from the Apollo/Apollo2 equivalents. There are essentially two primary transfer functions that can be used for half-duplex send (TX) or receive (RX) or full-duplex operation. A transfer can be called with either the blocking or nonblocking interface. The blocking interface returns after the transfer has been completed. The nonblocking interface returns after the transfer has been scheduled. There is an optional callback that can be supplied to the nonblocking interface to notify the application when the operation is complete.

SECTION 10

IOS HAL

The Apollo3 IOS HAL interface has been greatly simplified from Apollo/Apollo2. Support for direct LRAM interaction has been deprecated in preference to the FIFO interface.

10.1 Configuration

Configuration of the IOS device can be mapped as follows:

Table 10-1: IOS HAL Configuration Functional Map

Apollo/Apollo2 Function	Apollo3 Function
N/A	am_hal_ios_initialize
am_hal_ios_pwrctrl_enable am_hal_ios_pwrctrl_disable	am_hal_ios_power_ctrl
am_hal_ios_config	am_hal_ios_configure
N/A	am_hal_ios_control
N/A	am_hal_ios_uninitialize

10.2 Operation

Operation of the IOS device can be mapped as follows:

Table 10-2: IOS HAL Operation Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_ios_enable	am_hal_ios_enable
am_hal_ios_disable	am_hal_ios_disable

10.3 Interrupts

Interrupt control of the IOS device can be mapped as follows:

Table 10-3: IOS HAL Interrupts Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_ios_host_int_set am_hal_ios_host_int_clear am_hal_ios_host_int_get am_hal_ios_host_int_enable	am_hal_ios_control(AM_HAL_IOS_REQ_HOST_INT*)
	N/A
	N/A
	N/A
am_hal_ios_access_int_enable	N/A
am_hal_ios_access_int_enable_get	N/A
am_hal_ios_access_int_disable	N/A
am_hal_ios_access_int_clear	N/A
am_hal_ios_access_int_set	N/A
am_hal_ios_access_int_status_get	N/A
am_hal_ios_int_enable	am_hal_ios_interrupt_enable
am_hal_ios_int_disable	am_hal_ios_interupt_disable
am_hal_ios_int_clear	am_hal_ios_interrupt_clear
am_hal_ios_int_set	N/A
am_hal_ios_int_status_get	am_hal_ios_interrupt_status_get
am_hal_ios_fifo_service	am_hal_ios_interrupt_service

10.4 Data Movement

Table 10-4: IOS HAL Data Movement Functional Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_ios_fifo_space_left	am_hal_ios_fifo_space_left
am_hal_ios_fifo_space_used	am_hal_ios_fifo_space_used
am_hal_ios_fifo_write	am_hal_ios_fifo_write
am_hal_ios_fifo_write_simple	N/A
am_hal_ios_fifo_ptr_set	N/A
am_hal_ios_update_fifoctr	am_hal_ios_control(AM_HAL_IOS_REQ_FIFO_UPDATE_CTRL)
am_hal_ios_read_poll_complete	am_hal_ios_control(AM_HAL_IOS_REQ_READ_POLL)
am_hal_ios_lram_write	N/A

SECTION

11

PDM HAL

11.1 Configuration

The Apollo2 PDM HAL provided a number of macros to configure individual registers of the PDM block. These macros have been deprecated. In addition, the Apollo3 HAL follows the practice of using enums and booleans instead of discrete uint32_t for configuration parameters.

Table 11-1: PDM HAL Configuration Functional Map

Apollo2 Function	Apollo3 Function
N/A	am_hal_pdm_initialize
N/A	am_hal_pdm_power_control
am_hal_pdm_config	am_hal_pdm_configure
N/A	am_hal_pdm_deinitialize

11.2 Operation

Table 11-2: PDM HAL Operation Functional Map

Apollo2 Function	Apollo3 Function
am_hal_pdm_enable	am_hal_pdm_enable
am_hal_pdm_disable	am_hal_pdm_disable

11.3 Interrupts

Table 11-3: PDM HAL Interrupts Functional Map

Apollo2 Function	Apollo3 Function
am_hal_pdm_int_enable	am_hal_pdm_interrupt_enable
am_hal_pdm_int_disable	am_hal_pdm_interrupt_disable
am_hal_pdm_int_clear	am_hal_pdm_interrupt_clear
am_hal_pdm_int_status_get	am_hal_pdm_interrupt_status_get

11.4 Data Movement

Table 11-4: PDM HAL Data Movement Functional Map

Apollo2 Function	Apollo3 Function
am_hal_pdm_fifo_depth_read	N/A
am_hal_pdm_fifo_data_read	N/A
am_hal_pdm_fifo_flush	am_hal_pdm_fifo_flush
N/A	am_hal_pdm_dma_start

SECTION

12

PWRCTRL HAL

The Apollo3 SDK PWRCTRL is fairly consistent with Apollo/Apollo2 SDK. The biggest difference is the mapping of peripheral and memory configurations to enums as follows:

```
typedef enum
{
    AM_HAL_PWRCTRL_PERIPH_NONE,
    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_UART0,
    AM_HAL_PWRCTRL_PERIPH_UART1,
    AM_HAL_PWRCTRL_PERIPH_ADC,
    AM_HAL_PWRCTRL_PERIPH_SCARD,
    AM_HAL_PWRCTRL_PERIPH_MSPI,
    AM_HAL_PWRCTRL_PERIPH_PDM,
    AM_HAL_PWRCTRL_PERIPH_BLEL,
    AM_HAL_PWRCTRL_PERIPH_MAX
} am_hal_pwrctrl_periph_e;


typedef enum
{
    AM_HAL_PWRCTRL_MEM_NONE,
    AM_HAL_PWRCTRL_MEM_SRAM_8K_DTCM,
    AM_HAL_PWRCTRL_MEM_SRAM_32K_DTCM,
    AM_HAL_PWRCTRL_MEM_SRAM_64K_DTCM,
    AM_HAL_PWRCTRL_MEM_SRAM_96K,
    AM_HAL_PWRCTRL_MEM_SRAM_128K,
    AM_HAL_PWRCTRL_MEM_SRAM_160K,
    AM_HAL_PWRCTRL_MEM_SRAM_192K,
    AM_HAL_PWRCTRL_MEM_SRAM_224K,
    AM_HAL_PWRCTRL_MEM_SRAM_256K,
    AM_HAL_PWRCTRL_MEM_SRAM_288K,
    AM_HAL_PWRCTRL_MEM_SRAM_320K,
    AM_HAL_PWRCTRL_MEM_SRAM_352K,
```

```
    AM_HAL_PWRCTRL_MEM_SRAM_384K,
    AM_HAL_PWRCTRL_MEM_FLASH_512K,
    AM_HAL_PWRCTRL_MEM_FLASH_1M,
    AM_HAL_PWRCTRL_MEM_CACHE,
    AM_HAL_PWRCTRL_MEM_ALL,
    AM_HAL_PWRCTRL_MEM_MAX
} am_hal_pwrctrl_mem_e;
```

The function mappings are almost equivalent.

Table 12-1: PWRCTRL HAL Function Mappings

Apollo2 Function	Apollo3 Function
am_hal_pwrctrl_periph_enable	am_hal_pwrctrl_periph_enable
am_hal_pwrctrl_periph_disable	am_hal_pwrctrl_periph_disable
N/A	am_hal_pwrctrl_periph_enabled
am_hal_pwrctrl_memory_enable	am_hal_pwrctrl_memory_enable
am_hal_pwrctrl_bucks_init am_hal_pwrctrl_bucks_enable am_hal_pwrctrl_bucks_disable	N/A
am_hal_pwrctrl_low_power_init	am_hal_pwrctrl_low_power_init

SECTION

13

STIMER HAL

The Apollo3 SDK STIMER HAL is 100% compatible with Apollo2, except for the addition of the following functions to set an read back the 4 32-bit words of NVRAM.

Table 13-1: STIMER HAL Function Map

Apollo3 Function
am_hal_stimer_nvram_set
am_hal_stimer_nvram_get

SECTION

14

UART HAL

The Apollo3 SDK UART HAL greatly simplifies the HAL API from Apollo/Apollo2 HAL while deprecating the concept of string/char processing and embedding TX/RX buffering into the HAL.

14.1 Configuration

Table 14-1: UART HAL Configuration Function Map

Apollo/Apollo2 Function	Apollo3 Function
N/A	am_hal_uart_initialize
am_hal_uart_pwrctrl_enable am_hal_uart_pwrctrl_disable am_hal_uart_power_on_restore am_hal_uart_power_off_save	am_hal_uart_power_control
am_hal_uart_config am_hal_uart_clock_enable am_hal_uart_clock_disable am_hal_uart_fifo_config am_hal_uart_init_buffered	am_hal_uart_configure
N/A	am_hal_uart_deinitialize

14.2 Operation

Table 14-2: UART HAL Operation Function Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_uart_enable	N/A
am_hal_uart_disable	

14.3 Interrupts

Table 14-3: UART HAL Interrupts Function Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_uart_int_enable	am_hal_uart_interrupt_enable
am_hal_uart_int_disable	am_hal_uart_interrupt_disable
am_hal_uart_int_clear	am_hal_uart_interrupt_clear
am_hal_uart_int_status_get	am_hal_uart_interrupt_status_get
am_hal_uart_service_buffered am_hal_uart_service_buffered_timeout_save	am_hal_uart_interrupt_service
am_hal_uart_int_enable_get	N/A

14.4 Data Movement

Table 14-4: UART HAL Data Movement Function Map

Apollo/Apollo2 Function	Apollo3 Function
am_hal_uart_char_transmit_polled am_hal_uart_string_transmit_polled am_hal_uart_char_receive_polled am_hal_uart_line_receive_polled am_hal_uart_char_transmit_buffered am_hal_uart_string_transmit_buffered am_hal_uart_char_receive_buffered	am_hal_uart_transfer
N/A	am_hal_uart_tx_flush
am_hal_uart_flags_get am_hal_uart_status_get am_hal_uart_get_status_buffered	am_hal_uart_flags_get

6500 River Place Boulevard, Building 7, Suite 200, Austin, TX 78730
www.ambiq.com
sales@ambiq.com
+1 (512) 879-2850

A-SOCAP3-UGGA02EN v1.1
January 2023