DATASHEET ADDENDUM

Apollo2 SoC Flash Programming Helper Functions

Ultra-Low Power Apollo SoC Family

DA-A2-1p1

Legal Information and Disclaimers

Revision History

Revision	Date	Description
1.0	May 2018	Document initial public release
1.1	November 22, 2023	Updated document template

Reference Documents

Document ID	Description

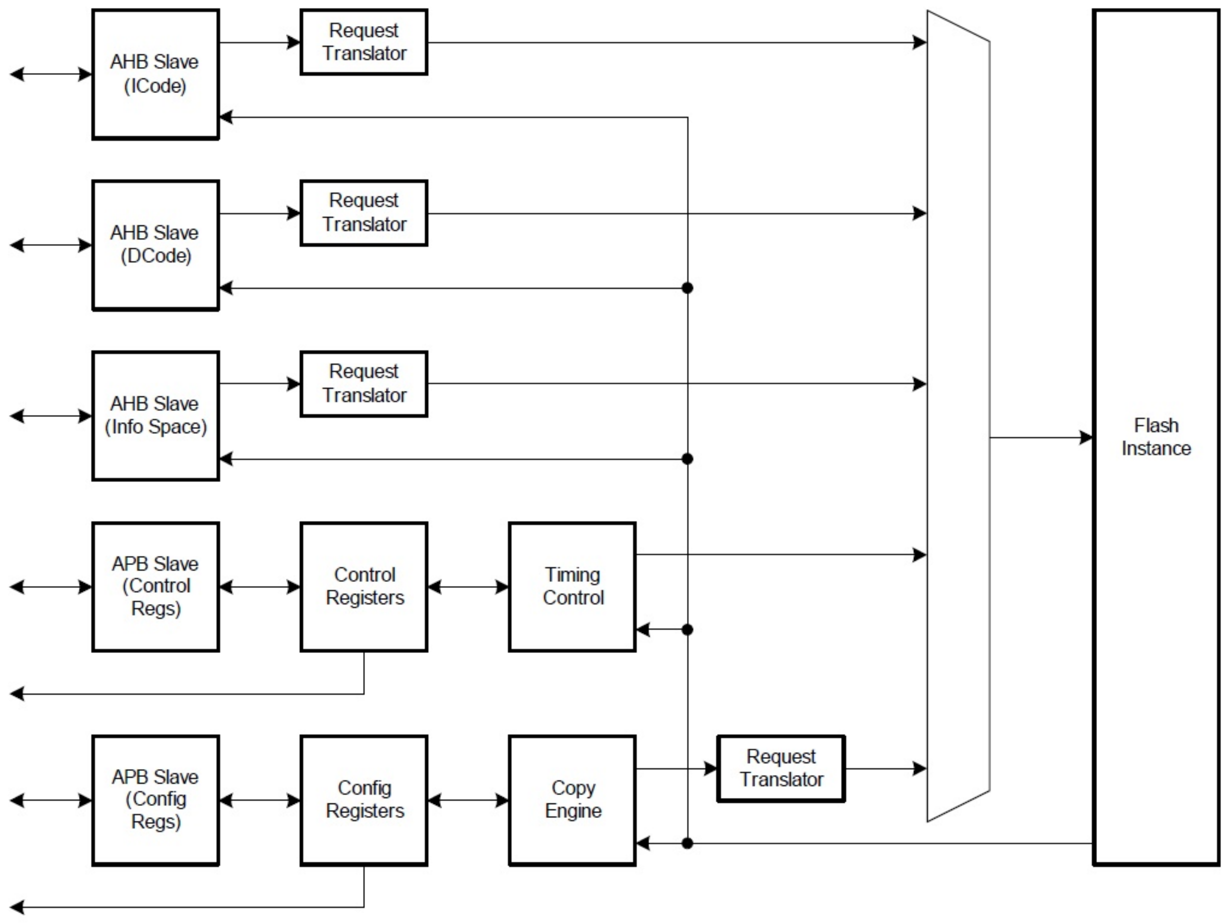
Table of Contents

SECTION

1

Flash Memory Controller

Figure 1-1: Block Diagram for the Flash Memory Controller



1.1 Functional Overview

During normal MCU code execution, the Flash Memory Controller translates requests from the CPU core (via the Flash cache) to the flash memory instance for instruction and data fetches. The Controller is designed to return data to the cache in single wait-state and can operate up to the maximum operating frequency of half the CPU core.

The Controller facilitates flash erase and programming operations through the control registers. When erase or programming operations are active, data cannot be fetched from the flash memory. This will be naturally handled by the cache controller fill logic to stall until the program operation is complete and the Flash device is available. With the cache enabled, this collision should happen very infrequently.

Another function of the Controller is to capture the configuration values which are distributed to the various on-chip peripherals of the SoC at chip power-up. These are read from the Information Space of the flash memory and captured in registers to be used by the other peripherals. The configuration values are reloaded each time a full-chip POI cycle occurs.

Figure 1-1 on page 5 shows a block diagram of the Flash Memory Controller.

SECTION

2 Directly Called Flash Helper Functions

The following flash helper functions are intended to be called directly from programs running either in flash or in SRAM. These are standard C functions that perform standard operations on either the flash or the INFO space.

In the following descriptions for Apollo2 SoC flash helper functions, the following definitions are used:

- **Instance**: Apollo2 SoC has two main flash instances, each being 512KB, and two INFO instances, each being one page or 8KB. The first main flash instance is addressed at 0 – 0x0007FFFF, the second at 0x00080000 – 0x000FFFFF.
- **Page**: Each Apollo2 SoC flash page size is 8KB for both main flash and each INFO instance. Therefore, one instance of main flash consists of 64 pages, or 128 pages total for both instances. Each instance of INFO space consists of a single page.

2.1 Directly Called Function to Erase an Instance of Main Flash

Use this function to erase the entire content of one instance (512 KB) of flash.

```
/****************************************//**
* @brief entry point for erasing one instance of main flash.
*
* This function operates on one flash instance per call.
*
* Calling this function erases an entire flash instance (512 KB).
*
* @param value PROGRAM key
* @param flash_inst 0 or 1 selects first or second flash block
* @return 0 for success, non-zero for failure
*
*******************************************/

#define PROGRAM_KEY (0x12344321)
```

```
int flash_mass_erase(
    uint32_t value,
           uint32_t flash_inst
    uint32_t flash_inst );
```

2.2 Directly Called Function to Erase one Page in the Main Flash

One page in the flash is 8192 bytes. This function can be used to erase a single page in the main flash.

```
/***************************************//**
* @brief entry point for erasing one page in the MAIN block of one
flash instance
*
* This function operates on one page of the MAIN block of one flash
instance per call.
*
* Calling this function erases one page in the MAIN block of one flash
instance (8192 bytes).
*
* @param value PROGRAM key
* @param flash_inst 0 or 1 selects first or second flash block
* For flash addresses 0 - 0x7FFFF, this parameter should be 0.
* For flash addresses 0x80000 - 0x100000, this parameter should be 1.
* @param PageNumber offset to a page aligned target location to begin
page erasing in the flash MAIN block
* @return 0 for success, non-zero for failure
*
*****************************************/
#define PROGRAM_KEY (0x12344321)
int flash_page_erase(
    uint32_t value,
    uint32_t flash_inst,
             uint32_t PageNumber );
```

2.3 Directly Called Function to Program N Words in the Customer INFO space

Use this function to write words in the customer INFO space such as the protection bits or other user

desired permanent information such as a unique ID.

```
/****************************************//**
* @brief entry point for programming the user portion (8192 bytes) of
the 16384 byte OTP.
*
* Calling this function programs multiple words in the OTP
*
* @param value Customer program key
* @param info_inst Typically set to zero.
* @param pSrc Pointer to word aligned array of data to program into
the flash INFO space.
* @param Offset Word offset in to the INFO space (0x01 means the sec-
ond 32 bit word)
* NOTE Offset == 0 --> first word in customer INFO space.
* @param NumberOfWords Number of words to program
* NOTE 0 < NumberOfWords < 2048 and (Offset + NumberOfWords) < 2048
* @return 0 for success, non-zero for failure
*
*******************************************/
#define PROGRAM_KEY (0x12344321)
int flash_program_info_area(
uint32_t value,
uint32_t info_inst,    // Typically 0
uint32_t *pSrc,
uint32_t Offset,       // 0 <= Offset < 256
uint32_t NumberOfWords // 1 < NumberOfWords < 257
);
```

2.4 Directly Called Function to Erase Customer INFO Space

Use this function to erase an instance of the customer INFO space.

```
/****************************************//**
* @brief entry point for erasing one instance of the customer INFO
block.
*
* This function operates on the customer INFO block on one FLASH
instance per call.
* Calling this function erases the customer INFO block on one flash
instance (8192 bytes).
*
* @param value Customer program key
* @param info_inst Typically set to zero (setting to 1 could invali-
date the part).
```

```
* @return 0 for success, non-zero for failure
*
******************************************/
* @brief Program N words of the MAIN block of one flash instance
*
* Calling this function programs N words of the MAIN block of 1 or
more flash instances.
* Note that programming can cross main flash instances, therefore the
entire 1MB of main
* flash can be programmed with a single call to this function.
*
* @param value Customer program key (PROGRAM_KEY)
* @param pSrc Pointer to word aligned data to program into the flash
MAIN block.
* WARNING: pSrc cannot point back in to flash itself
* @param pDst pointer to word aligned target location to begin pro-
gramming in the flash MAIN block
* @param NumberOfWords Number of 32-bit words to program
*
* NOTE 0 < NumberOfWords < 2048 and (pDst + (NumberOfWords*4)) < 1MB
*
* @return 0 for success, non-zero for failure
*
*
******************************************/
#define PROGRAM_KEY (0x12344321)
int flash_program_main(
    uint32_t value,
    uint32_t *pSrc,
    uint32_t *pDst,
    uint32_t NumberOfWords);
```

2.5 Directly Called Function to Erase Main Flash and Customer INFO Space

Use this function to erase an instance of main flash AND customer INFO space.

```
/****************************************//**
* @brief entry point for erasing one instance of the customer INFO
space.
*
* This function operates on the main flash and the customer INFO
space.
* Calling this function erases the customer INFO block on one flash
instance (8192 bytes).
*
* @param value Customer program key
* @param instance 0 or 1.
* @return 0 for success, non-zero for failure
*
******************************************/
```

```
#define PROGRAM_KEY (0x12344321)
int flash_info_plus_main_erase(
    uint32_t value,
    uint32_t instance ); // Typically 0
```

2.6 Directly Called Function to Erase ALL Main Flash and Customer INFO Space

Use this function to erase the entirety of main flash as well as all of the customer INFO space.

```
/****************************************//**
* @brief entry point for erasing both customer INFO instances and
both main flash instances.
*
* This function operates on the main flash and the customer INFO
space.
* Calling this function erases the customer INFO block on one flash
instance (8192 bytes).
*
* @param value Customer program key
* @return 0 for success, non-zero for failure
*
*******************************************/
#define PROGRAM_KEY (0x12344321)
int flash_info_plus_main_erase_both(
    uint32_t value);
```

SECTION

3

Debugger Invoked Helper Functions

The following functions are intended to be invoked from an external debugger or debugger surrogate such as a manufacturing gang programmer. The debugger sets up a few parameters in SRAM and copies the binary data to be programmed in to the flash in to the SRAM behind the parameters. It then sets the PC to the entry point for this function and resumes execution from there. When the function completes, it hits a hard coded break point instruction which passes control back to the debugger or debugger surrogate.

3.1 Program Words in the Flash from a Debugger

The debugger uses this function to copy binary data from SRAM to a target location in flash.

```
/****************************************//**
* @brief entry point for reading program details and data from SRAM
and then programming it into MAIN
*
* Calling this function looks up programming parameters starting at
offset 0x0 in SRAM
* 0x10000000 pointer in to flash
* 0x10000004 number of 32-bit words to program
* 0x10000008 PROGRAM key to pass to flash helper routine
* 0x1000000C return code debugger sets this to -1 all RCs are >= 0
* 0x10001000 first 32-bit word of data buffer to be programmed
(WRITE_BUFFER_START)
* Note this routine hits a break point instruction to transfer con-
trol to a debugger surrogate in the parallel programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
******************************************/
#define PROGRAM_KEY (0x12344321)
#define WRITE_BUFFER_START (0x10001000)
void flash_program_main_from_sram(void);
```

3.2 Program Words in the INFO Space from a Debugger

The debugger uses this function to copy binary data from SRAM to a target location in the INFO area.

```
/*****************************************//**
* @brief entry point for reading program details and data from SRAM
and then programming it into the OTP
*
* Calling this function looks up programming parameters starting at
offset 0x0 in SRAM
* 0x10000000 Word offset in to the INFO space, 0 <= Offset < 256
* 0x10000004 Instance number, 0 or 1 (typically 0)
* 0x10000008 Number of 32-bit words to program
* 0x1000000C Program key to pass to flash helper routine
* 0x10000010 Return code. Debugger sets this to -1, all RCs are >= 0
* 0x10001000 first 32-bit word of data buffer to be programmed
(WRITE_BUFFER_START)
*
* Note this routine hits a break point instruction to transfer con-
trol to a debugger surrogate in the parallel programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
*
*******************************************/
#define PROGRAM_KEY (0x12344321)
void flash_program_info_area_from_sram(void);
```

3.3 Erase a Number of Contiguous Main Flash Pages from a Debugger

The debugger uses this function to erase a number of contiguous pages in main flash.

```
/*****************************************//**
* @brief entry point for erasing a contiguous block of main block
pages from SRAM
*
* Calling this function looks up page erase information from offset
0x0 in SRAM
* 0x10000000 Instance number, 0 or 1.
* If the first page number is located in the flash address range of
* For 0 <= PageNumber <= 63 (addresses 0 - 0x7FFFF), this parameter
should be 0.
* For 64 <= PageNumber <= 127 (addresses 0x80000 - 0x100000), this
parameter should be 1.
* 0x10000004 Number of main flash pages to erasemust be between 1 and
128 inclusive
* 0x10000008 PROGRAM key to pass to flash helper routine
* 0x1000000C Return code, debugger sets this to -1, all RCs are >= 0
* 0x10000010 PageNumber of the first flash page to erase, 0 <=
PageNumber <= 127.
* Note - an error is returned if PageNumber + number_of_pages > 128.
* Note this routine hits a break point instruction to transfer con-
trol to a debugger surrogate in the parallel programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
*
*
******************************************/
#define PROGRAM_KEY (0x12344321)
void flash_erase_main_pages_from_sram(void);
```

3.4 Erase an Entire Flash Instance From a Debugger

The debugger uses this function to entire flash instance.

```
/****************************************//**
* @brief entry point for reading mass erase parameters from SRAM and
then erasing the main block
*
* Calling this function looks up programming information from offset
0x0 in SRAM
* 0x10000000 Instance number
* For flash instance addresses 0 - 0x7FFFF, this parameter should be
0.
* For flash instance addresses 0x80000 - 0xFFFFF, this parameter
should be 1.
* 0x10000004 Program key to pass to flash helper routine
* 0x10000008 Return code, debugger sets this to -1, all RCs are >= 0
* Note this routine spins when flash_mass_erase() returns and waits
for the debugger surrogate in the parallel
programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
*
******************************************/
#define PROGRAM_KEY (0x12344321)
void flash_mass_erase_from_sram(void);
```

3.5 Erase an Entire INFO Instance From a Debugger

The debugger uses this function to an entire customer INFO instance.

```
/****************************************//**
* @brief entry point for reading mass erase parameters from SRAM and
then erasing an entire
* (customer) INFO instance.
*
```

```
* Calling this function looks up programming information from offset
0x0 in SRAM
* 0x10000000 INFO instance number, 0 or 1.
* 0x10000004 Program key to pass to flash helper routine
* 0x10000008 Return code, debugger sets this to -1, all RCs are >= 0
*
* Note this routine spins when flash_info_erase() returns and waits
for the debugger surrogate in the parallel
programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
*
*
*******************************************/
#define PROGRAM_KEY (0x12344321)
void flash_info_erase_from_sram(void);
```

3.6 Erase an Entire Flash Main and INFO Instance From a Debugger

The debugger uses this function to entire flash AND an entire (customer) INFO instance.

```
/****************************************//**
* @brief entry point for reading mass erase parameters from SRAM and
then erasing an entire the main instance
* and an entire (customer) INFO instance.
*
* Calling this function looks up programming information from offset
0x0 in SRAM
* 0x10000000 Instance number, 0 or 1.
* For flash instance addresses 0 – 0x7FFFF, this parameter should be
0.
* For flash instance addresses 0x80000 – 0xFFFFF, this parameter
should be 1.
* 0x10000004 Program key to pass to flash helper routine
* 0x10000008 Return code, debugger sets this to -1, all RCs are >= 0
*
* Note this routine spins when flash_mass_erase() returns and waits
for the debugger surrogate in the parallel
programmer.
*
* @return never returns, spins here waiting for debugger or debugger
surrogate on the parallel programmer
*
*
*******************************************/
#define PROGRAM_KEY (0x12344321)
void flash_info_plus_main_erase_from_sram(void);
```

3.7 Special Purpose Functions

The following functions perform special functions. Each of these functions have a specific purpose which is explained on a function-by-function basis.

3.7.1 Read a Word From a Given Memory Address

This special-purpose function can be used to safely read a memory location no matter the state of the flash. It can therefore be used to safely read INFO space, peripheral registers, etc.

```
/****************************************//**
* @brief entry point for safely reading one at a specified address.
*
* @param pSrc Address at which to read a word.
*
*******************************************/
int flash_util_read_word(uint32_t *pSrc);
```

3.7.2 Write a Word To a Given Location

This special-purpose function can be used to safely write a memory location no matter the state of the flash. It can therefore be used to safely write SRAM, peripheral registers, and more.

```
/****************************************//**
* @brief entry point for safely reading one at a specified address.
*
* @param pDest Address at which to write a word.
* @param ui32Value Value to be written.
*
*******************************************/
int flash_util_write_word(uint32_t *pDst, uint32_t ui32Value);
//##### INTERNAL BEGIN #####
```

3.7.3 Customer Flash Recovery

This special-purpose function can be used to recover flash and customer INFO space.

```
/****************************************//**
*
* @brief entry point for customer recovery of a part.
*
* This function clears the customer info spaces and main blocks of
both flash instances and wipes SRAM.
*
* Calling this function erases the INFO block on both flash instances
plus their MAIN blocks.
```

```
* In addition, it simultaneously wipes sram and finishes by forcing a
POI reset.
* This function will succeed even if the customer has INFO protection
bits set.
* It will not succeed if any AMBIQ internal protection bits are set.
* The contents of a secure boot loader in flash instance 0 will be
left intact.
*
* @param value     customer brick key
*
* @return          0 for success, non-zero for failure
*
******************************************/
#define BRICK_KEY (0xA35C9B6D)
void flash_recovery( uint32_t value);
//##### INTERNAL END #####
```

3.8 Non-Blocking Functions

The following functions perform functions in a non-blocking fashion. The calling program can start an operation and come back later to determine whether it completed or not.

3.8.1 Non-Blocking Operation Complete

```
/****************************************//**
*
* @brief entry point for determining if non-blocking flash operation
has completed.
*
* Calling this function returns the busy state of the flash control-
ler
*
* @return 0 for busy and non-zero for flash controller finished.
*
******************************************/
bool flash_nb_operation_complete();
```

3.8.2 Non-Blocking Mass Erase Instance

```
/****************************************//**
*
* @brief entry point for non-blocking erase of one instance
*
* This function operates on one FLASH instance per call.
*
* Calling this function erases an entire flash instance (512KB).
*
* @param value customer program key
* @param flash_inst 0 or 1 selects first or second flash block
```

```
*
* @return 0 for success, non-zero for failure
*
*****************************************/
int flash_mass_erase_nb(uint32_t value, uint32_t flash_inst);
```

3.8.3 Non-Blocking Page Erase

```
/***************************************//**
*
* @brief entry point for non-blocking erase of one page.
*
* This function operates on one page of the MAIN FLASH instance per
call.
*
* Calling this function erases one page in the MAIN block of one flash
instance (8192 bytes).
*
* @param value customer program key
* @param flash_inst 0 or 1 selects first or second flash instance
* @param PageNumber offset to a page aligned target location to begin
page erasing in the FLASH
MAIN block
*
* @return 0 for success, non-zero for failure
*
*****************************************/
int flash_page_erase_nb(uint32_t value, uint32_t flash_inst,
uint32_t PageNumber);
```

SECTION 4

HAL Support for the Flash Helper Functions

The AmbiqSuite firmware Hardware Abstraction Layer (HAL) provides routines for finding and accessing all of the directly called flash helper functions. See the latest **am_hal_flash.c** and **am_hal_flash.h** modules for more information.

The HAL contains a function pointer table to gain access to the flash helper functions which looks like this:

```
//
*****************************************************************************
//
// Structure of function pointers to helper functions for invoking various
// flash operations. The functions we are pointing to here are in the Apollo2
// integrated BOOTROM.
//
//
*****************************************************************************
typedef struct am_hal_flash_helper_struct
{
   //
   // The basics.
   //
   int (*flash_mass_erase)(uint32_t, uint32_t);
   int (*flash_page_erase)(uint32_t, uint32_t, uint32_t);
   int (*flash_program_main)(uint32_t, uint32_t *,
                       uint32_t*, uint32_t);
   int (*flash_program_info)(uint32_t, uint32_t,
                       uint32_t*, uint32_t, uint32_t);

   //
   // Non-blocking variants, but be careful these are not interrupt safe so
   // mask interrupts while these very long operations proceed.
   //
   int (*flash_mass_erase_nb)(uint32_t, uint32_t);
   int (*flash_page_erase_nb)(uint32_t, uint32_t, uint32_t);
   bool (*flash_nb_operation_complete)(void);
```

```
    //
    // Essentially these are recovery options.
    //
    int (*flash_erase_info)(uint32_t, uint32_t);
    int (*flash_erase_main_plus_info)(uint32_t, uint32_t);
    int (*flash_erase_main_plus_info_both_instances)(uint32_t);
    void (*flash_recovery)(uint32_t);

    //
    // Useful utilities.
    //
    uint32_t (*flash_util_read_word)(uint32_t*);
    void (*flash_util_write_word)(uint32_t*, uint32_t);
    void (*delay_cycles)(uint32_t);

    //
    // The following functions pointers will generally never be called from
    // user programs. They are here primarily to document these entry points
    // which are usable from a debugger or debugger script.
    //
    void (*flash_program_main_sram)(void);
    void (*flash_program_info_sram)(void);
    void (*flash_erase_main_pages_sram)(void);
    void (*flash_mass_erase_sram)(void);
    void (*flash_erase_info_sram)(void);
    void (*flash_erase_main_plus_info_sram)(void);
} g_am_hal_flash_t;
```

Following is an example of the HAL abstraction of the flash helper functions:

```
//
*****************************************************************************
//
//! @brief This programs up to N words of the Main array on one flash instance.
//!
//! @param ui32Value - The programming key, AM_HAL_FLASH_PROGRAM_KEY.
//! @param pui32Src - Pointer to word aligned array of data to program into
//! the flash.
//! @param pui32Dst - Pointer to the word aligned flash location where
//! programming of the flash is to begin.
//! @param ui32NumWords - The number of words to be programmed.
//!
//! This function will program multiple words in main flash.
//!
//! @return 0 for success, non-zero for failure.
//
//
*****************************************************************************
int
am_hal_flash_program_main(uint32_t ui32Value, uint32_t *pui32Src,
             uint32_t *pui32Dst, uint32_t ui32NumWords)
```

Finally, a simple example of calling the HAL mass erase function.

```
int32_t i32ReturnCode = am_hal_flash_mass_erase(AM_HAL_FLASH_PROGRAM_KEY, 1);
//
// Check for an error from the HAL.
//
if (i32ReturnCode)
{
    am_util_stdio_printf("FLASH_MASS_ERASE i32ReturnCode = 0x%x.\n",
                         i32ReturnCode);
    i32ErrorFlag++;
}
    else
{
    am_util_stdio_printf("FLASH_MASS_ERASE successful\n");
}
```

DA-A2-1p1
November 2023