White Paper

Alif MCU RAM Regions and Linker Files

Version 1.1

Table of Contents

Introduction

Alif MCUs in the Ensemble family contain up to 13.5 MB of total internal SRAM. Through the global memory map, all SRAM banks are accessible by any master in the system. This document will describe the SRAM banks and their differences so that they can be best utilized by software.

RAM Regions

Below is a table describing the size, address location, and operating frequency of each SRAM bank in Ensemble Gen 2 devices. These are the maximum sizes that can be available. The actual size and availability of each SRAM bank depend on the part number of the Ensemble device. You may refer to the part number decoder in the product's datasheet for more information on SRAM sizing options.

Gen 2 Silicon

Name	Global Address	Size [KB]	Clock [MHz]
SRAM0	0x02000000	4096	400
SRAM1	0x08000000	2560	400
SRAM2	0x50000000	256	400
SRAM3	0x50800000	1024	400
SRAM4	0x58000000	256	160
SRAM5	0x58800000	256	160
SRAM6	0x62000000	2048	160
SRAM7	0x63000000	512	160
SRAM8	0x63100000	2048	160
SRAM9	0x60000000	768	160

SRAM0 and SRAM1

These two SRAM banks reside in the high-performance region of the MCU. They are intended to be used as bulk SRAM, i.e. general-purpose memory by one or more cores. The memory can be used privately by a single core, or it can be used for shared data intended to be readable by multiple entities. The output framebuffers for an LCD panel or input framebuffers for a Camera module would be placed in these banks, for example.

The two SRAMs run at 200MHz each in Rev A silicon, and at 400MHz each in Rev B silicon, served by a 400MHz main bus. With this approach, one core or DMA can utilize the full bandwidth of one SRAM bank and another core or DMA can utilize the full bandwidth of the second SRAM bank with no degradation of performance. This is useful in a multi-core and multi-DMA system.

SRAM2, SRAM3, SRAM4, and SRAM5

These four SRAM banks are not as general-purpose as the SRAM0 and SRAM1 banks. Rather, SRAM2 through SRAM5 operate as the Tightly Coupled Memory (TCM) for the M55 cores in the RTSS-HP and RTSS-HE subsystems. The TCM is a high-bandwidth and low-latency memory that is primarily used by the M55 core it is attached to.

Two of the banks, SRAM2 and SRAM3, are clocked at the same frequency as the RTSS-HP domain and operate as the M55-HP core's high-speed ITCM and DTCM, respectively. The other two banks, SRAM4 and SRAM5, are clocked at the same frequency as the RTSS-HE domain and operate as the M55-HE core's high-speed ITCM and DTCM, respectively.

The TCM access time is a single clock cycle with no wait states. So interrupt routines, general code, and data can be processed with minimum latency when located in the TCM. There is a single bus to access the ITCM, typically used for instruction memory, and four independent buses to access the DTCM, typically used for data memory. This multiple bus architecture is meant to facilitate Single Instruction Multiple Data (SIMD) or vector operations.

Each M55 core sees its own ITCM memory mapped at 0x00000000, and its DTCM mapped at 0x20000000. A core cannot use its own TCMs global addresses. Utility functions LocalToGlobal and GlobalToLocal handle the address mapping for when a potentially-TCM address needs to be passed to another part of the system.

Since the TCM is accessible through the global address map it is possible to configure peripherals and DMAs to read/write data to/from the TCM. This feature is meant to eliminate the need to copy data between "working memory" close to the CPU and slower memory for other systems to access. A peripheral like I2S or the camera controller can use the DMA to place audio data or image data directly in the high-speed TCM and notify the CPU that it is ready to process.

SRAM6, SRAM7, SRAM8, SRAM9

These two SRAM banks reside in the high-efficiency region of the MCU. Just like SRAM0 and SRAM1, they are intended to be used as low-power bulk SRAM. Although the banks are general-purpose, they are also lower clocked than the high-performance SRAM banks so a small performance hit will be observed.

L1 Caches

Beside each Cortex-M55 core there is 32kB of L1 Instruction cache and 32kB of L1 Data cache. The caches are fully hardware managed based on configuration given to the MPU. Using the MPU, your software can assign caching policies to one or more memory address ranges which then affect the caching behavior of those addresses.

Through the MPU configuration, code and data can be placed in fast L1 Cache after being read from a slower memory location such as MRAM or SRAM. All subsequent reads of this code and data will come from cache rather than from slower memory to improve overall performance. This is called "read-allocate" cache policy. Data written to MRAM, or SRAM can also be stored in cache first allowing the core to continue with other operations rather than waiting for the write operation to complete in the target memory region. The data written, even if the destination address is in the global memory map, will stay in the data cache until an instruction is used to invalidate the data cache. This is called "write-back with write-allocate" policy and cache invalidation functions are needed to push data out from

cache to the actual memory. If data is meant to be always written to MRAM or SRAM directly, then you would use a “write-through” cache policy instead.

There is also a “transient” hint bit in the memory attributes – this hints that the data is unlikely to be re-read. Transient data is prioritized for eviction when new data needs to be cached. It makes sense to set this attribute for memory holding large frame buffers, which are typically processed in a linear fashion, so that any given part won’t be revisited until the next pass – it prevents the travel through the buffer from evicting more general program data.

Prefetch Unit

The M55 has an automatic prefetch unit that identifies linear access patterns by monitoring cache misses and preloads data into the cache. This means that the access latency of the SRAM can be hidden in many “data streaming” cases. So care should be taken to process large SRAM buffers in a linear fashion to activate this.

It is also possible to insert manual prefetch hints into code using the __builtin_prefetch function. This will not gain anything if the automatic prefetch unit has been successful.

TCM

When the TCM of a core is accessed through its local address (0x0 for ITCM or 0x20000000 for DTCM), then the access is single-cycle, and no caching is involved; cache-related MPU attributes are ignored. When the TCM of another core is accessed through global memory addressing then the MPU caching behavior is applied. It is important to set the write-through bits as well as the read and write allocation bits appropriately.

Memory Coherency

There is no hardware cache coherency between the M55 cores and other bus masters. If a memory region is being accessed by an M55 core and another master, that region must either be marked non-cacheable or shareable, or the cache must be cleaned and/or invalidated before or after the other master accesses the region.

(If sharing a TCM, only the remote core needs to worry about caches – the local core always performs fast uncached accesses to it.)

Non-cached access is very slow, so it’s generally best to mark regions at least write-through cacheable, and perform the maintenance operations as necessary. If a region is write-back cached (but not write-through), you must clean the relevant address range after writing and before another master reads. It is always necessary to invalidate the cache before reading data written by another master.

Invalidate-only D-cache operations should be used with care, as they can potentially lose data that has only been written to the cache. Use only when you are certain that the range in question has nothing that needs to be written-back. It is almost never safe to perform a global D-cache invalidate in normal operation.

Note that the M55 can perform speculative reads to normal memory, so memory could be cached despite the code not having explicitly read it. Therefore invalidate operations must be made after the other master has finished writing.

Tip: Ranged clean and/or invalidate operations can be slow if on a large range of addresses, due to the need to loop through those addresses. If the range that needs to be worked on is larger than around

128KiB, it can be more time-effective to perform a global clean or clean+invalidate operation, looping through only 32KiB of cache sets and ways.

Arm MPU Configuration example

In the mpu_table[], we have SRAM0, SRAM6, SRAM8, and MRAM regions defined and assigned to memory attribute 1. We also have SRAM1 region defined and assigned to memory attribute 2. Regions under attribute 1 have read and write cache allocation enabled as well as write-back mode enabled. This is best for general purpose code and data. Regions under attribute 2 only have read-allocation enabled and write-through mode is in use, together with the transient hint.

```
static void MPU_Load_Regions(void)
{
    static const ARM_MPU_Region_t mpu_table[] __STARTUP_RO_DATA_ATTRIBUTE = {
        {
            .RBAR = ARM_MPU_RBAR(0x02000000UL, ARM_MPU_SH_NON, 0UL, 1UL, 0UL),    // RO, NP, XN
            .RLAR = ARM_MPU_RLAR(0x023FFFFFUL, 1UL)     // SRAM0
        },
        {
            .RBAR = ARM_MPU_RBAR(0x08000000UL, ARM_MPU_SH_NON, 0UL, 1UL, 0UL),    // RO, NP, XN
            .RLAR = ARM_MPU_RLAR(0x0827FFFFUL, 2UL)     // SRAM1
        },
        {
            .RBAR = ARM_MPU_RBAR(0x70000000UL, ARM_MPU_SH_NON, 0UL, 1UL, 1UL),
            .RLAR = ARM_MPU_RLAR(0x71FFFFFFUL, 0UL)     // LP- Peripheral & PINMUX Regions */
        },
        {
            .RBAR = ARM_MPU_RBAR(0x62000000UL, ARM_MPU_SH_NON, 0UL, 1UL, 0UL),    // RO, NP, XN
            .RLAR = ARM_MPU_RLAR(0x621FFFFFUL, 1UL)     // SRAM6
        },
        {
            .RBAR = ARM_MPU_RBAR(0x63100000UL, ARM_MPU_SH_NON, 0UL, 1UL, 0UL),    // RO, NP, XN
            .RLAR = ARM_MPU_RLAR(0x632FFFFFUL, 1UL)     // SRAM8
        },
        {
            .RBAR = ARM_MPU_RBAR(0x80000000UL, ARM_MPU_SH_NON, 1UL, 1UL, 0UL),    // RO, NP, XN
            .RLAR = ARM_MPU_RLAR(0x8057FFFFUL, 1UL)     // MRAM
        },
    };

    /* Define the possible Attribute regions */
    ARM_MPU_SetMemAttr(0UL, ARM_MPU_ATTR_DEVICE);    /* Device Memory */
    ARM_MPU_SetMemAttr(1UL, ARM_MPU_ATTR(    /* Normal Memory, Write-back, Read/Write-Allocate */
                            ARM_MPU_ATTR_MEMORY_(1,1,1,1),
                            ARM_MPU_ATTR_MEMORY_(1,1,1,1)));
    ARM_MPU_SetMemAttr(2UL, ARM_MPU_ATTR(    /* Normal Memory, Transient, Write-through, Read-Allocate */
                            ARM_MPU_ATTR_MEMORY_(0,0,1,0),
                            ARM_MPU_ATTR_MEMORY_(0,0,1,0)));

    /* Load the regions from the table */
    ARM_MPU_Load(0U, &mpu_table[0], sizeof(mpu_table)/sizeof(ARM_MPU_Region_t));
}
```

Note that there are default attributes for memory addresses not covered by the loaded table – see *Chapter B8: The System Address Map* in the *Armv8-M Architecture Reference Manual*. These attributes would make most Alif SRAM regions cacheable.

Memory Retention

Some SRAMs in the high-efficiency domain of the SoC have low leakage properties allowing our MCU to offer memory retention modes. When the MCU enters low power STANDBY or STOP Mode, the SRAM banks can be configured to remain on and retain their contents while the rest of the MCU powers down.

The optional retentions in Rev A silicon are as follows:

- Option 1: 512 KB is retained by enabling the option on SRAM2 and SRAM3.
- Option 2: 2048 KB is retained by enabling the option on SRAM6.

The optional retentions in Rev B silicon are as follows:

- Option 1: 256 KB is retained by enabling the option on half of SRAM2 and half of SRAM3.
- Option 2: 512 KB is retained by enabling the option on SRAM2 and SRAM3 entirely.
- Option 3: 1920 KB is retained by enabling the option on SRAM6 and SRAM7 (subset of total)

Reasoning within Use Cases

The following section describes why one memory region is more suitable than another, depending on the usage.

Use Case 1:

M55 core is processing a stream of data

- Bad choice: process data in SRAM or use M55 to fetch data
- Ideal choice: use DMA to place data in to M55's TCM

Why: The M55 core's access time to TCM is a single clock cycle with no wait states. If the M55 needs to fetch data from slower memory regions, then this has an impact to performance. The M55 core's L1 cache helps to reduce accesses to slower memory regions containing code or data in other use cases but this use case involves processing a stream of incoming data that cannot be cached. Since the TCM is accessible through the global address map it is possible to configure peripherals and DMAs to read/write data to/from the TCM.

Another reason to place data in TCM is to take advantage of the multiple bus architecture when performing Single Instruction Multiple Data (SIMD) or vector operations. With four independent buses to DTCM and one to ITCM, it is possible to load a SIMD instruction and up to 2 data parameters in a single cycle.

Use Case 2:

Extremely large double-buffer graphics use case.

- Bad choice: application is loaded to SRAM or TCM – leaves no room for framebuffer
- Ideal choice: application should XIP from MRAM and rely on M55 caching

Why: The M55 core's L1 cache is very efficient at prefetching data from MRAM. There is very little performance degradation shown in our internal benchmarking with cache enabled. Using XIP means the application code and read-only data remains stored in MRAM and read-write data is stored in TCM. This configuration leaves 100% of the bulk SRAM available for use as the graphics framebuffer.

Use Case 3:

Small ML model.

- Good choice: model in MRAM, activation buffer in SRAM
- Faster choice: model in MRAM, activation buffer in nearest DTCM

Why: The model data is large, read-only and the Ethos-U55 makes relatively few accesses to it – it works perfectly well as XIP from MRAM. The activation buffer has far more random read and write accesses, and its speed significantly affects the overall inference time. If the buffer can fit in the DTCM of the neighboring M55, then the lower access latency can significantly boost performance. A 40% speed-up was observed doing this for keyword spotting in the high-efficiency subsystem.

Doing this requires version 22.11 or later of the Ethos-U core driver, as the ethosu_address_remap hook must be implemented to handle conversion of local TCM addresses that the software sees to global addresses that the Ethos-U55 sees.

Use Case 4:

In a multi-core example, one can set up a pipeline operation between two cores. In this use case, one core is collecting data from sensors and then applying a pre-processing algorithm on the data collected. Then, another core can access the processed data from the first core via the global TCM interface. Doing so will minimize data copy and support ping-pong buffering.

Code Examples

The following code example describes how to define a buffer and assign it a section name. This should be the same between compilers.

```
uint8_t byte_buffer[BUFFER_SIZE] __attribute__((used, section("section_name")));
```

Note: the "used" keyword tells the linker file not to remove this section during any optimization steps.

Scatter Example

How to use an Arm Clang scatter file to place a named section into a specific memory region.

Below is an example of section names "lcd_frame_buf" and "camera_frame_buf" placed into SRAM banks 0 and 1, respectively.

```
RW_SRAM0 SRAM0_BASE SRAM0_SIZE  {
 * (lcd_frame_buf)                ; LCD frame Buffer
}
RW_SRAM1 SRAM1_BASE SRAM1_SIZE  {
 * (camera_frame_buf)             ; Camera Frame Buffer
}
```

Linker Example

Below is a GNU-style Linker example of section names "lcd_frame_buf" and "camera_frame_buf" placed into SRAM banks 0 and 1, respectively. The NOLOAD keyword is used to tell the linker that these read/write regions are zero initialized and therefore do not need to be part of the application binary.

```
.bss.at_sram0 (NOLOAD) : ALIGN(8)
{
  * (.bss.lcd_image_buf)                /* LCD Frame Buffer */
} > SRAM0

.bss.at_sram1 (NOLOAD) : ALIGN(8)
{
  * (.bss.camera_frame_buf)             /* Camera Frame Buffer */
} > SRAM1
```

A zero table is defined in the linker file as well. Using these zero table entries will make sure that the memory region will be zeroed-out before use. That operation is part of the init code which runs before main function starts.

```
.zero.table :
{
  __zero_table_start__ = .;
  LONG (ADDR(.bss))
  LONG (SIZEOF(.bss)/4)
  LONG (ADDR(.bss.at_sram0))
  LONG (SIZEOF(.bss.at_sram0)/4)
```

```
    LONG (ADDR(.bss.at_sram1))
    LONG (SIZEOF(.bss.at_sram1)/4)
    __zero_table_end__ = .;
  } > MRAM
```

Further Reading

Alif Processor System Reference Manual (login required):
https://swrm.alifsemi.com/
Arm Cortex-M55 Technical Reference Manual (public):
https://developer.arm.com/documentation/101051/0101/
Arm Cortex-M v8 Architecture Reference Manual
https://developer.arm.com/documentation/ddi0553/latest/

Document History

Version	Change Log
1.0	Initial public release covering Rev A beta silicon and Rev B production silicon
1.1	Revised for Gen 2 production silicon