CODE TIME TECHNOLOGIES

mAbassi RTOS

BSP Document

ARMv8 Caches (GCC)

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document details the L1, L2 and L3 caches, memory management unit (MMU) support BSP module for the multi-core mAbassi RTOS. This module is targeted to the ARM v8 multi-core processor, more specifically the Arm53 MPcore, using GCC

1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

Table 1-1 Distribution

File Name	Description
`ARMv8_SMP_L1_L2_GCC.s`	Cache configuration and enabling code

1.2 Limitations

The file `ARMv8_SMP_L1_L2_GCC.s`, described here, can only be use with the multi-core RTOS mAbassi. For the single core Abassi, the file `ARMv8_L1_L2_GCC.s` must be used [R2].

1.3 Features

This cache BSP module handles the configuration and enabling of the MPcore L1, L2 and L3 (when present) cache, and the memory management unit (MMU). All four (4) levels of MMU tables are supported meaning the granularity of control for the type of caching of the address space can be configured on segments as small as 4096 bytes.

2 Target Set-up

All there is to do to configure and enable the ARMv8 caches is to include in the build the file `ARMv8_SMP_L1_L2_GC.s`. either in a makefile or with the development tool GUI.

2.1 Build Options & Tables

The file `ARMv8_SMP_L1_L2_GCC.s` relies on a few build options for its configuration and some definition tables for setting up of the L1, L2 and L3 caches and the MMU. The build options are listed in the following table:

Table 2-1 Build options

Build Option	Description
`OS_N_CORE`	Number of cores the application uses
`OS_PLATFORM`	Specifies the target platform
`OS_SAME_L1_PAGE_TBL`	Select if a single MMU set of page table is used or if each core has its own MMU table. The name is kept the same as the ARMv7.
`OS_MMU_ALL_INVALID`	Select if the MMU sets the unused pages as invalid or shared
`OS_MMU_EXTERN_DEF`	Select if the MMU definition tables are imported or local
`OS_MMU_TBL_2`	Number of level #2 tables
`OS_MMU_TBL_3`	Number of level #3 tables
`OS_NC_STRONG_ORDER`	Select if non-cache memory has the “device” or “strongly ordered” property

The cache configuration module internally uses three definition tables. One of the tables defines the memory blocks that are configured as cached/shared amongst all the cores. Another table defines where are located the peripherals in the memory space as it is necessary to bypass the caches when accessing peripherals; this table if for non-cached memory. A third table defines the virtual and physical addresses of the private memory area of each core (non-shared memory). These tables are described in more details in section 2.3.

2.2 Build Options

All build options can be set (overloaded) through the command line. Using a fictitious build option `OS_BUILD_OPTION`, the default value assigned to `OS_BUILD_OPTION` in `ARMv8_SMP_L1_L2_GCC.s` can be overloaded by using the assembler command line option `--defsym` and specifying the new value (`1234`), as shown in the following example:

Table 2-2 Command line set of `OS_BUILD_OPTON` (ASM)

```
aarch64-none-eabi-as  … --defsym OS_BUILD_OPTION=1234  …
```

The default value of `OS_BUILD_OPTION` could be overloaded if the compiler is used to assemble the file. In the following example, the new value is set to `6789`:

Table 2-3 Command line set of `OS_BUILD_OPTION` (C)

```
aarch64-none-eabi-gcc  … -D OS_BUILD_OPTON=6789 …
```

All default build options are set as show on this table and can be directly changed by editing the `.equ` value in `ARMv8_SMP_L1_L2_GCC.s`:

Table 2-4 `OS_BUILD_OPTION` modification

```
#ifndef OS_BUILD_OPTION
 .ifndef OS_BUILD_OPTION
   .equ  OS_BUILD_OPTION,        32          // Build option example
 .endif
#endif
```

2.2.1 Number of cores

When operating the mAbassi RTOS on a platform, the RTOS needs to be configured for the number of cores it has access to, or will use. This number is most of the time the same as the number of cores the device has, but it can also be set to a value less than the total number of cores on the device, but not larger obviously. This must be done for both the `mAbassi.c` file and the `ARMv8_SMP_L1_L2_GCC.s` file, through the setting of the build option `OS_N_CORE`. In the case of the file `mAbassi.c`, `OS_N_CORE` is one of the standard build options. If `OS_N_CORE` is not specified, the default value used depends on the target platform defined by the build option `OS_PLATFORM` (next section)

2.2.2 Target Device

Each device/platform has its own memory and peripheral mapping. As such, the valid memory ranges are quite likely different between targets. The default MMU tables are set based on the value assigned to the token `OS_PLATFORM`. At the time of writing this document, the following platforms are supported:

Table 2-5 OS_PLATFORM valid settings

Target Platform	`OS_PLATFORM` value
Xilinx / UltraScale+	`0x7753`

If in the future there are platforms that are not listed in the above table, the numerical values assigned to the platform are specified in comments in the file `ARMV8_SMP_L1_L2_GCC.s`., right beside the internal definition of `OS_PLATFORM`.

2.2.3 Number of L1 Page Table(s)

The L1 Cache and MMU use a 4 levels of page tables to hold the information of the caching and sharing characteristics all memory area, and that table also holds the translation information from virtual to

physical memory. If the whole useable memory map is shared, then there is no need to use one table per core, as the individual tables are exactly the same. But if there are some memory areas that are defined as private to a core, then each core must have its own private table. The value assigned to the build option `OS_SAME_L1_PAGE_TBL` controls if each core has its own table or if all cores share a single one. To give each core its own table (this is the default setting), the build option `OS_SAME_L1_PAGE_TBL` must be set to a value of zero (0). To share all the tables amongst all the cores the build option `OS_SAME_L1_PAGE_TBL` must be set to a non-zero value.

NOTE: If the build option `OS_SAME_L1_PAGE_TBL` is set to a positive value, meaning to use a single L1 page table for all cores, and both non-shared tables are not filled with all zeros, the problem is trapped during run-time and the cache initialization code enters an infinite loop.

2.2.4 Page Tables

The Cache and MMU use 4 levels of page tables to hold the information of the caching and sharing characteristics of each 4 KB pages of the total memory range, and these tables also hold the translation information from virtual to physical memory. If the whole useable memory map is shared (same for all cores), then there is no need to use one set of table per core, as the individual tables are exactly the same. But if there are some memory areas that are defined as private to a core, then each core must have its own private table. The value assigned to the build option `OS_SAME_L1_PAGE_TBL` controls if each core has its own table or if all cores share a single one. To give each core its own table (this is the default setting), the build option `OS_SAME_L1_PAGE_TBL` must be set to a value of zero (0). To share the same table amongst all the cores, the build option `OS_SAME_L1_PAGE_TBL` must be set to a non-zero value. If the private definition table specified private memory regions and `OS_SAME_L1_PAGE_TBL` is set to a non-zero value, the problem is trapped at run time and an infinite loop is entered. By looking at the comments on the right of the infinite loop instruction indicated the need to use individual set of table per cores.

2.2.5 MMU Definition Tables

The MMU tables are filled using the definition tables (see Section 2.3 for more information on the definition tables). By default, the definition tables are located in the file `ARMv8_SMP_L1_L2_GCC.s`. By setting the build option `OS_MMU_EXTERN_DEF` to a non-zero value, the definition tables are imported from outside the file `ARMv8_SMP_L1_L2_GCC.s`.

When the definition tables are imported, the imported variables replace the tables defined by the following labels:

Table 2-6 OS_MMU_EXTERN_DEF synbols

Table Label	Imported Symbol
SharedInfo	G_MMUsharedTbl
NonCacheInfo	G_MMUnonCachedTbl
PrivateInfo	G_MMUprivateTbl
NonCprivInfo	G_MMUnonCprivTbl

The way the imported table must be constructed is exactly the same as the internal tables are constructed (see Section 2.3). Reusing the example in section 2.3.2 for the peripheral definition table, the imported table should be like:

Table 2-7 Example peripheral definition table

```
int64_t G_MMUnonCachedTbl[] = {0x0000000000001000, 0x00000000E0000000,
                               0x0000000001000000, 0x0000000FF0000000,
                               0x0000000000000000 };
```

2.2.6 Unused Pages

The unused pages (the pages that are not mapped to a peripheral (non-cached) area, nor mapped as a shared memory area, and not mapped in the non-shared (private) area) can be tagged as being either invalid to provokes an abort when read/written, or they can be tagged as valid cached/shared. The value assigned to the build option `OS_MMU_ALL_INVALID` controls if the unused memory areas are set as invalid or as shared. To set the unused memory as shared (this is the default setting), the build option `OS_MMU_ALL_INVALID` must be set to a value of zero (0). To set the unused memory as invalid, the build option `OS_MMU_ALL_INVALID` must be set to a non-zero value.

2.2.7 Number of Level 2 and 3 tables

The MMU tables can traverse 4 levels of table: from level 0 to 3. The number of level 0 and level 1 tables is always the same, but the number of level 2 and level 3 required depends entirely on memory mapping used in the MMU definition tables. By default, the maximum number of level 2 and level 3 is set to 32 for each level and for each core. If 32 tables are insufficient, the problem is trapped during run-time and the cache initialization code enters an infinite loop. Comments in uppercase letters on the right of the infinite loop indicates which table level is undersized. To modify the maximum number of level 2 tables, set the build option `OS_MMU_TBL_2` to a larger value than 32. If it's the number of level 3 tables that is undersize, set the build option `OS_MMU_TBL_3` to a larger value than 32.

2.2.8 Non-cache attribute

Non-cache accesses can be set with one of two attributes: "Device" or "Strongly Ordered". On the ARMv8, the documentation states both attributes are the same at the CPU level; the difference between the two been on the AXI bus `AxCACHE` values. "Device" sets the `AxCACHE` value to `0001b` and "Strongly Ordered" sets the `AxCACHE` values to `0000b`.

The non-cache access attribute is set by the value assigned to the build option `OS_NC_STRONG_ORDER`. This is a bit field, where bit #0 specifies the attribute for the shared non-cache accesses and bit #1 specifies the attributes for the private (non-shared) non-cached accesses. A bit clear to 0 selects the access as "Device", and when set to 1 it selects the accesses to "Strongly Ordered". The default value of `OS_NC_STRONG_ORDER` is 0, specifying all non-cache accesses attributes to be "Device".

2.2.9 ARM Cache Errata

None of the current ARM v8 Cache errata affect the Cache BSP or mAbassi code.

2.3 Tables

The cache configuration module uses internal definition tables to specify the characteristics of different memory areas. Three tables are used:

- Shared / cached memory
- Shared / un-cached memory (typically for peripherals)
- Private memory (non-shared) / cached memory
- Private memory (non-shared) / uncached memory

At around line 100 in the file in `ARMv8_SMP_L1_L2_GCC.s`, one will find a commented-out block of statements that shows how these three tables are set-up. This is a detailed example and is not used. Past around line 120, the real default tables are defined. There is one set of table per known platform. These sets of table are the ones used and should be modified as required by the target application, or overloaded by imported them by setting the build option `OS_MMU_EXTERN_DEF` (Sect 2.2.5) to a non-zero value.

2.3.1 Shared Memory

The shared memory definition table contains all the information needed to program the MMU page table to select the address space to be cached and shared amongst all the cores. The way the table is constructed is through the use of pairs of numerical values. These pairs specify the base address of the memory block and the address range of the memory block. There can be as many pairs as the system requires, and cached / shared memory blocks don't need to be in contiguous memory pages.

The table is referenced by the label `SharedInfo`. The first entry in a pair is the number of bytes the memory block spans and the second value in a pair is the base address of memory block. The table must be terminated with a size of 0.

Table 2-8 Example shared memory definition table

```
SharedInfo:                          // Table to define the shared/cached memory areas
@ ----------- SIZE ----- ADDRESS -- // when the whole table sdefaults to invalid
   .8byte  0x00100000, 0x02000000    // The table holds pairs of value (size, base) and
   .8byte  0                         // is terminated with a size of 0
```

2.3.2 Peripheral Addresses

The peripheral memory definition table contains all the information needed to program the page table to make the address space occupied by the peripherals a non-cached shared area. The way the table is constructed is through the use of pairs of numerical values. These pairs specify the base address of the peripherals and the address range of peripheral registers. There can be as many pairs as the system requires, as peripherals registers addresses do not need to be in contiguous memory pages. For example, a system could have a peripheral where its registers are located between addresses `0xE0000000` and `0xE0000FFF` and all the rest of the peripheral registers are located between addresses `0xFF000000` and `0xFFFFFFFF`.

The table is referenced by the label `PeriphInfo`. The first entry in a pair is the number of bytes the block of peripheral register spans and the second value in a pair is the base address of the peripheral register block. The table must be terminated with a size of 0.

The example peripheral definition table is located around line 90 in the file `ARMv8_SMP_L1_L2_GCC.s`. Taking the simple example described above, the table should be filled as follows:

Table 2-9 Example peripheral definition table

```
NonCacheInfo:
@ ----------- SIZE ----- ADDRESS ---------  // Table to define the non-cached /
peripheral areas
   .8byte  0x00001000, 0xE0000000        // Size of the page / Base address of the page
   .8byte  0x01000000, 0xFF000000
   .8byte  0                             // Size = 0 is the end of the definition table
```

Although the minimum Cache / MMU memory pages are 4KB (`0x00001000`), if the specified size for a block of peripheral registers is not an exact multiple of `0x00001000`, it will be internally set at the `ceil` value to the next exact multiple of 4KB. In the case of the base address, if the specified address for a block of registers is not an exact multiple of `0x00001000`, it will be internally set at the floored value to the next exact multiple of 4KB. It is strongly advised to always specify the sizes and addresses in exact multiple of 4 KB as a page crossing with a size too small will end up doing an incorrect set-up. An example of a bad definition is to set a size of `0x00000100` for a base address of `0x8FFFFFF0`.

2.3.3 Private Memory

The private memory definition table is used to program the page table to make the address space assigned as private to a core; this means memory pages that are cached but non-shared and non-coherent. The way the table is constructed is through the use of triplets of numerical values. These triplets specify the virtual base address of a private memory section (the memory pages base address the core sees), the physical base address of that private memory section (the base address in the physical memory), and the size of the private memory section. There can be as many triplets as the application requires, therefore the different private sections of memory do not need to be contiguous.

As an example, let's take a 4-core device and the application requires each core to have 16Mbyte of private memory each. Assuming the private memory is seen by each at address `0x80000000`, then 4 physical blocks of 16Mbyte of memory are required; one block per core. The 4 blocks of physical memory reserved for the private memory are located at addresses `0x80000000`, `0x81000000`, `0x82000000` and `0x83000000`). Once the page table is configured, none of the cores will have access to the memory located between addresses `0x81000000` to `0x83FFFFFF` and each core will have its own private memory between the addresses `0x80000000` and `0x80FFFFFF`.

There are two tables for non-shared memory: the one defined at label The table is referenced by the label `PrivateInfo`. Specifies the private (re-mapped) memory regions with caching and the one at lable `NonCprivInfo` specifies the private (re-mapped) memory region wit no caching. Both tbales use the same structure, therefore the explanations provided for the cached / non-shared regions `PrivateInfo` with apply to the non-cached / non-shared `NonCprivInfo` also.

The first entry in a triplet is the number of bytes the block of private memory spans. The second value in a triplet is the virtual base address of the private memory block. The third entry is the base address of the physical memory attached to that private memory block. The table must be terminated with a size of 0 for each core and there must be at least as many zero-size terminated groups as there are cores in the application (as defined by `OS_N_CORE` Section 2.2.1).

The example for the private memory definition table is located around line 150 in the file `ARMv8_SMP_L1_L2_GCC.s`.

Taking the example described above, the table should be filled as follows:

Table 2-10 Example private memory definition table

```
PrivateInfo:                          // Table to define the core private memory area
// --------- SIZE ------ V ADDR -- PH ADDR -  // Terminated with 0 Size.
  .8byte  0x01000000, 0x80000000, 0x80000000  // Triplets: Size / Virt addr / Phys addr
  .8byte  0                     // Core #0 Non-shared: 0x80000000 mapped to 0x80000000

  .8byte  0x01000000, 0x80000000, 0x81000000
  .8byte  0                     // Core #1 Non-shared: 0x80000000 mapped to 0x81000000

  .8byte  0x01000000, 0x80000000, 0x82000000
  .8byte  0                     // Core #2 Non-shared: 0x80000000 mapped to 0x82000000

  .8byte  0x01000000, 0x80000000, 0x83000000
  .8byte  0                     // Core #3 Non-shared: 0x80000000 mapped to 0x83000000
```

2.3.4 Overlapping Regions

If the definition tables have by mistake being set with overlapping regions, the final caching property of the overlapped region is set according to the order the MMU tables are created. If memory regions (physical addresses for shared, virtual addresses for non-shrared) overlap amongst the 4 type of memory accesses, the last pass is the one that sets the final MMU table entry:

- First pass: Shared / cached
- Second pass: Shared / non-cached
- Third pass: Non-Shared / cached
- Final pass: Non-Shared / non-cached

For example, if a shared / cached memory region overlap the same non-shared / cached virtual memory region, because the non-shared / cached is the last pass to fill that MMU table entry, the final property is non-shared / cached.

3 Implementation

The cache configuration performs the following operations in this listed order:

1) Disable SMP.
2) Invalidation of the L1
3) Invalidation of L2 and L3 caches.
4) Enable SMP
5) The page table set-up for the whole memory as default (either invalid or cached and shared)
6) The page table entries are set-up for the specified shared memory (cached)
7) The page table entries are set-up for the specified shared memory (non-cached)
8) The page table entries for the current core specified private memory / cached are set up
9) The page table entries for the current core specified private memory / non-cached are set up
10) The SCU is set-up and enabled
11) The MMU is set-up and enabled
12) The Instruction and Data caches are enabled

When a single L1 MMU table is selected, configured with the build option `OS_SAME_L1_PAGE_TBL` set to a non-zero value (Sect 2.2.3), only core #0 performs steps 3), 5), 6) and 7), and as a single L1 MMU table is selected, steps 8) and 9) are do-nothing steps as both non-shared tables have to be filled with 0s.

4 API

A single function (`COREcacheON()`) is supplied in this distribution. The following sub-section describes the `COREcacheON()` function.

4.1 COREcacheON

Synopsis

```
#include "mAbassi.h"

void COREcacheON (void);
```

Description

`COREcacheON()` is the module used to configure and enable the L1 and L2 caches, and the MMU. The build option values needed to set-up the cache as required by the application are described earlier in this document.

Availability

Optional module.

Arguments

`void`

Returns

`void`

Component type

Function

Options

Notes

See also

4.2 DCacheFlushRange

Synopsis

```
#include "mAbassi.h"

DCacheFlushRange(void *Addr, int Len);
```

Description

`DCacheFlushRange()` is used to flush (write the cache contents in the external RAM) a range of data addresses from the L1 and L2 cache. The base address to flush is specified with the argument `Addr` and the number of consecutive addresses to flush is specified by the argument `Len`. As the caches lines are 64 bytes and because only full cache linescan be flushed, the real start address is the exact multiple by 64 bits address that is lower or equal to the argument `Addr`. This also mean the real final address will be the address with an exact multiple of 64 minus 1, that is greater or equal to (`Addr + Len -1`).

Availability

Optional module.

Arguments

`Addr:`	Start address (lower) of the memory range to flush from the L1 and L2 cache
`Len:`	Number of consecutive addresses of the memory range to flush from the L1 and L2 cache.

Returns

`void`

Component type

Function

Options

Notes

See also

`DCacheInvalRange()`	Section 4.3

4.3 DCacheInvalRange

Synopsis

```
#include "mAbassi.h"

DCacheInvalRange(void *Addr, int Len);
```

Description

`DCacheInvalRange()` is used to invalidate (to force a re-reading the external memory) a range of data addresses from the L1 and L2 cache. The base address to invalidate is specified with the argument `Addr` and the number of consecutive addresses to invalidate is specified by the argument `Len`. As the caches lines are 64 bytes and because only full cache lines can be invalidated, the real start address is the exact multiple by 64 bits address that is lower or equal to the argument `Addr`. This also mean the real final address will be the address with an exact multiple of 64 minus 1, that is greater or equal to (`Addr + Len -1`).

Availability

Optional module.

Arguments

`Addr:`	Start address (lower) of the memory range to invalidate from the L1 and L2 cache
`Len:`	Number of consecutive addresses of the memory range to invalidate from the L1 and L2 cache.

Returns

`void`

Component type

Function

Options

Notes

See also

`DCacheFlushRange()`	Section 4.2

4.4 MMUlog2Phy

Synopsis

```
#include “mAbassi.h”

void *MMUlog2Phy(const void *Addr);
```

Description

`MMUlog2phy()` is used report the physical address associated to a logical address. This information is needed when non-shared memory must be accessed externally from the processor.

Availability

Optional module.

Arguments

`Addr:`	Logical address to report the physical location

Returns

`void *`	Physical address

Component type

Function

Options

Notes

See also

`MMUphy2Log()`	Section 4.5

4.5 MMUphy2Log

Synopsis

```
#include "mAbassi.h"

void *MMUphy2Log(const void *Addr);
```

Description

`MMUphy2Log()` is used report the logical address associated to a physcial address. This information could be useful with non-shared.

Availability

Optional module.

Arguments

`Addr:`	Physical address to report the physical location

Returns

`void *`	Logical address

Component type

Function

Options

Notes

See also

`MMUlog2Phy()`	Section 4.4

5 References

[R1] mAbassi Port – SMP ARM Cortex A53, available at http://www.code-time.com
[R2] Abassi RTOS – User Guide, available at http://www.code-time.com
[R3] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R4] Abassi – ARMv8 Caches