CODE TIME TECHNOLOGIES

mAbassi RTOS

BSP Document

ARMv7 Caches (CCS)

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document details the L1 and L2 caches, memory management unit (MMU), and snoop control unit (SCU) support BSP module for the multi-core mAbassi RTOS. This module is targeted to the ARM Cortex-A9 multi-core processor, commonly known as the Arm9 MPcore, using Texas Intruments' Code Composer Studio (CCS). The CCS used for test and validation is version 6.0.1 with compiler and assemble version 5.1.10.

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
`ARMv7_SMP_L1_L2_CCS.s`	Cache configuration and enabling code

1.2 Limitations

The file `ARMv7_SMP_L1_L2_CCS.s`, described here, can only be use with the multi-core RTOS mAbassi. For the single core Abassi, the file `ARMv7_L1_L2_CCS.s` must be used [R3].

1.3 Features

This cache BSP module handles the configuration and enabling of the MPcore L1 instruction and data caches, the memory management unit (MMU), the snoop control unit (SCU), and the ARM CoreLink Level 2 Cache Controller L2C-310.

2 Target Set-up

All there is to do to configure and enable the ARMv7 caches is to include in the build the file `ARMv7_SMP_L1_L2_CCS.s` and to make sure the port-specific assembly file build option `OS_USE_CACHE` is set to a non-zero value (refer to the mAbassi port document for CCS [R1]).

2.1 Build Options & Tables

The file `ARMv7_SMP_L1_L2_CCS.s` relies on a few build options for its configuration and some definition tables for setting up of the L1 caches and 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_ASM_THUMB`	Use Thumb2 or 32 bit instructions
`OS_PLATFORM`	Specifies the target platform
`OS_SAME_L1_PAGE_TBL`	Select if a single L1 MMU page table is used or if each core has its own MMU table
`OS_MMU_ALL_INVALID`	Select if the MMU sets the unused L1 pages as invalid or shared
`OS_MMU_EXTERN_DEF`	Select if the MMU definition tables are imported or local
`OS_L1_CACHE_BP`	Enable/disable the L1 cache branch prediction option
`OS_L1_CACHE_PF`	Enable/disable the L1 cache data pre-fetch option
`OS_CACHE_WRITE_ZERO`	Enable/disable the full line of zero write line option
`OS_USE_NON_SHARED`	Select if the cores have access to exclusive (private) memory areas
`OS_L2_BASE_ADDR`	Base address of the L2C-310 L2 cache registers / disable of the L2 cache

As a quick reference, the following table lists the combinations of values for the two build options that control the enabling or disabling of the two levels of caches:

Table 2-2 Enabling / Disabling combinations

Build Option value	L1 Cache / MMU / SCU	L2 Cache
`OS_SAME_L1_PAGE_TBL == -1` `OS_L2_BASE_ADDR == 0`	Disable	Disable
`OS_SAME_L1_PAGE_TBL != -1` `OS_L2_BASE_ADDR == 0`	Enable	Disable

`OS_SAME_L1_PAGE_TBL == -1`
`OS_L2_BASE_ADDR != 0`
`or OS_L2_BASE_ADDR undefined`
	Disable	Enable

`OS_SAME_L1_PAGE_TBL != -1`
`OS_L2_BASE_ADDR != 0`
`or OS_L2_BASE_ADDR undefined`
	Enable	Enable

The cache configuration module internally uses two or 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. A third table, when the feature is enabled, defines the virtual and physical addresses of the private memory area of each core (non-shared memory).

2.2 Build Options

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 `ARMv7_SMP_L1_L2_CCS.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. In the case of the file `ARMv7_SMP_L1_L2_CCS.s`, to modify the number of cores, all there is to do is to change the numerical value associated to the token definition of `OS_N_CORE`, located around line 40; this is shown in the following table. By default, the number of cores is set to 2.

Table 2-3 `OS_N_CORE` modification

```
  .if !($$defined(OS_N_CORE))
OS_N_CORE     .equ        2
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_N_CORE` in `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the required number of cores, as shown in the following example where the number of cores is set to 4:

Table 2-4 Command line set of `OS_N_CORE (ASM/C)`

```
armcl  … -D OS_N_CORE=4  …
```

Exactly the same value of `OS_N_CORE` as specified for the assembler must be specified for the compiler with the application "C" files.

The built option `OS_N_CORE` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-1 GUI setting of `OS_N_CORE`

2.2.2 Thumb2

The cache configuration file (`ARMv7_SMP_L1_L2_CCS.s`) is by default using 32-bit ARM instructions. The build option `OS_ASM_THUMB` (new in version 1.66.66) can be set to use Thumb2 instructions instead. The use of Thumb2 instruction is enabled when the build option `OS_ASM_THUMB` is set to a non-zero; by default, the token `OS_ASM_THUMB` is set to a non-zero value, enabling the special handling. As for other tokens, the numerical value associated to the `OS_ASM_THUMB` token, located around line 75, can be changed as shown in the following table:

Table 2-5 `OS_ASM_THUMB` modification

```
.if !($$defined(OS_ASM_THUMB))
OS_ASM_THUMB      .equ        1
  .endif
```

It is also possible to overload the `OS_ASM_THUMB` value set in `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the required base register index as shown in the following example:

Table 2-6 Command line set of `OS_ASM_THUMB` (ASM)

```
armcl  … -D OS_ASM_THUMB=1 …
```

NOTE: Never use the `--code-state=16` command line option with the `ARMv7_SMP_L1_L2_CCS.s` file.

The built option `OS_ASM_THUMB` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-2 GUI setting of `OS_ASM_THUMB`

2.2.3 Target Device

Each device has their own memory and peripheral mapping. As such, the valid memory ranges are quite likely different and the L2 cache registers, which are not based on ARM's peripheral base address, must be known in order to be able to configure the L2 cache. The useable memory areas and the L2 register base address are set by relying on the value assigned to the token `OS_PLATFORM`. At the time of writing this document, the following platforms are supported:

Table 2-7 OS_PLATFORM valid settings

Target Platform	`OS_PLATFORM` value
Altera / Cyclone V Soc FPGA	`0xAAC5`
Freescale / iMX6	`0xFEE6`
Texas Instruments / OMAP 4460	`0x4460`
Xilinx / Zynq XC7Z020	`0x7020`

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 `ARMV7_SMP_L1_L2_CCS.s`, right beside the internal definition of `OS_PLATFORM` (around line 45).

To select the target platform, all there is to do is to change the numerical value associated with the token `OS_PLATFORM` located around line 45 in the file `ARMV7_SMP_L1_L2_CCS.s`. By default, the target platform is the OMAP 4460, therefore `OS_PLATFORM` is assigned the numerical value `0x4460`. The following table shows how to set the target platform to the Freescale / iMX6, which is assigned the numerical value `0xFEE6`:

Table 2-8 `OS_PLATFORM` modification

```
  .if !($$defined(OS_PLATFORM))
OS_PLATFORM   .equ        0xFEE6
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_PLATFORM` in the file `ARMV7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the target platform numerical value:

Table 2-9 Command line set of `OS_PLATFORM`

```
armcl  … -D OS_PLATFORM=0xFEE6  …
```

The built option `OS_PLATFORM` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-3 GUI setting of `OS_PLATFORM`

2.2.4 L1 Page Table(s) / L1 disabling

The L1 Cache and MMU use a 16Kbyte page table to hold the information of the caching and sharing characteristics of each 1Mbyte page of the 4Gbyte total memory range, 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 the same table amongst all the cores, the build option `OS_SAME_L1_PAGE_TBL` must be set to a non-zero value and non-minus one (-1) value.

The build option `OS_SAME_L1_PAGE_TBL` can also be used to disable the L1 caches (data and instruction), the MMU and SCU. Setting the build option to a value of minus one (-1) does not configure, nor enable the L1 caches, the MMU, and the Snoop Control Unit (SCU).

The default setting is shown in the following table and this section of code is located around line 50 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-10 `OS_SAME_L1_PAGE_TBL` setting

```
  .if !($$defined(OS_SAME_L1_PAGE_TBL))
OS_SAME_L1_PAGE_TBL   .equ        0
  .endif
```

To use a single table for all cores, the build option `OS_SAME_L1_PAGE_TBL` must be set a positive value as shown in the following table:

Table 2-11 Using a single L1 page table

```
  .if !($$defined(OS_SAME_L1_PAGE_TBL))
OS_SAME_L1_PAGE_TBL   .equ        1
  .endif
```

To disable the L1 caches (instruction and data), MMU and SCU, the build option `OS_SAME_L1_PAGE_TBL` must be set to minus one (-1) as shown in the following table:

Table 2-12 Disabling the L1 cache / MMU / SCU

```
  .if !($$defined(OS_SAME_L1_PAGE_TBL))
OS_SAME_L1_PAGE_TBL   .equ        -1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_SAME_L1_PAGE_TBL` in the file `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example where the L1 caches are disabled:

Table 2-13 Command line set of `OS_SAME_L1_PAGE_TBL`

```
armcl  … -D OS_SAME_L1_PAGE_TBL=-1  …
```

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 the build option `OS_USE_NON_SHARED` (see Section 2.2.11) is set to a non-zero value indicating non-shared (private) memory areas are defined, an error will be issued during assembly, as it not possible to use the same L1 page table when the different cores have their own private memory sections.

The built option `OS_SAME_L1_PAGE_TBL` can also be set through CCS GUI in the *CCS Build → ARM Compiler → Advance Options → Predefined Symbols* menu as shown in the following figure:

Figure 2-4 GUI setting of `OS_SAME_L1_PAGE_TBL`

2.2.5 MMU Definition Tables

The MMU L1 tables are filled using definition tables (see Section 2.3 for more information on the definition tables). By default, the definition tables are located in the file `ARMv7_SMP_L1_L2_CCS.s`. By defining, and by setting the build option `OS_MMU_EXTERN_DEF` to a non-zero value, the definition tables are imported from outside the file `ARMv7_SMP_L1_L2_CCS.s`.

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

Table 2-14 OS_MMU_EXTERN_DEF synbols

Table Label	Imported Symbol
SharedInfo	`G_MMUsharedTbl`
NonCacheInfo	`G_MMUnonCachedTbl`
PrivateInfo	`G_MMUprivateTbl`

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 2-15 Example peripheral definition table

```
int G_MMUnonCachedTbl[] = {0x00001000, 0xE0000000,
                           0x01000000, 0xFF000000,
                           0x00000000 };
```

The data type of the imported definition table can be either `int`, or `void *`, or any pointers type, as long as the `sizeof()` of the data type selected is 4 bytes.

The default setting is shown in the following table, and this section of code is located around line 55 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-16 `OS_MMU_EXTERN_DEF` setting

```
  .if !($$defined(OS_MMU_EXTERN_DEF))
OS_MMU_EXTERN_DEF    .equ       0
  .endif
```

The default setting can be changed importing the tables; this is shown in the next table:

Table 2-17 `OS_MMU_EXTERN_DEF` for invalid setting

```
  .if !($$defined(OS_MMU_EXTERN_DEF))
OS_MMU_EXTERN_DEF    .equ       1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_MMU_EXTERN_DEF` in the file `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example:

Table 2-18 Command line set of `OS_MMU_EXTERN_DEF`

```
armcl  … -D OS_MMU_EXTERN_DEF =1  …
```

The built option `OS_MMU_EXTERN_DEF` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-5 GUI setting of `OS_MMU_EXTERN_DEF`

2.2.6 Unused Pages

The unused L1 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, which 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.

The default setting is shown in the following table, and this section of code is located around line 55 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-19 `OS_MMU_ALL_INVALID` setting

```
  .if !($$defined(OS_MMU_ALL_INVALID))
OS_MMU_ALL_INVALID    .equ       0
  .endif
```

The default setting can be changed to tag all unused memory areas as invalid memory; this is shown in the next table:

Table 2-20 `OS_MMU_ALL_INVALID` for invalid setting

```
  .if !($$defined(OS_MMU_ALL_INVALID))
OS_MMU_ALL_INVALID    .equ       1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_MMU_ALL_INVALID` in the file `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example:

Table 2-21 Command line set of `OS_MMU_ALL_INVALID`

```
armcl  … -D OS_MMU_ALL_INVALID=1  …
```

The built option `OS_MMU_ALL_INVALID` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-6 GUI setting of `OS_MMU_ALL_INVALID`

2.2.7 L1 Cache Branch Prediction

The L1 Cache can be set to use or to not use branch prediction. The value assigned to the build option `OS_L1_CACHE_BP` controls if branch prediction is used or not. To not use branch prediction (this is the default setting), the build option `OS_L1_CACHE_BP` must be set to a value of zero (0). To enable branch prediction, the build option `OS_L1_CACHE_BP` must be set to a non-zero value.

The default setting is shown in the following table, and this section of code is located around line 60 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-22 `OS_L1_CACHE_BP` setting

```
  .if !($$defined(OS_L1_CACHE_BP))
OS_L1_CACHE_BP    .equ        0
  .endif
```

The default setting can be changed to enable the branch prediction; this is shown in the next table:

Table 2-23 Enabling branch prediction

```
  .if !($$defined(OS_L1_CACHE_BP))
OS_L1_CACHE_BP    .equ        1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_L1_CACHE_BP` in the file `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example to enable L1 cache branch prediction:

Table 2-24 Command line set of `OS_L1_CACHE_BP`

```
armcl  … -D OS_L1_CACHE_BP=1  …
```

The built option `OS_SAME_L1_CACHE_BP`can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-7 GUI setting of `OS_SAME_L1_CACHE_BP`

2.2.8 L1 / L2 Cache Pre-fetch

The L1 and L2 Cache can be set to use or to not use pre-fetch. The values assigned to the build option `OS_L1_CACHE_PF` and `OS_L2_CACHE_PF` control if pre-fetching is used or not. To not use pre-fetching (this is the default setting), the build option `OS_L1_CACHE_PF` / `OS_L2_CACHE_PF` must be set to a value of zero (0). To enable pre-fetching, the build option `OS_L1_CACHE_PF` / `OS_L2_CACHE_PF` must be set to a non-zero. L1 and L2 pre-fetching is independently controlled.The default setting is shown in the following table, and this section of code is located around line 65 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-25 `OS_L1_CACHE_PF` setting

```
  .if !($$defined(OS_L1_CACHE_PF))
OS_L1_CACHE_PF       .equ       0
  .endif
```

The default setting can be changed to enable the data pre-fetch; this is shown in the next table:

Table 2-26 Enabling data pre-fetch

```
  .if !($$defined(OS_L1_CACHE_PF))
OS_L1_CACHE_PF       .equ       1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_L1_CACHE_BP` in the file `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example to enable L1 cache pre-fetching:

Table 2-27 Command line set of `OS_L1_CACHE_PF`

```
armcl  … -D OS_L1_CACHE_PF=1  …
```

The built option `OS_SAME_L1_CACHE_PF` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-8 GUI setting of `OS_SAME_L1_CACHE_PF`

2.2.9 Full line of write zero

When using both the L1 and L2 caches, it is possible to activate a mechanism that sends information from the L1 cache to the L2 cache when a cache line full of zero is written from the L1 cache to the L2 cache. This feature helps speed-up operations alike `memset()` with a fill value of zero. The value assigned to the build option `OS_CACHE_WRITE_ZERO` controls if the feature is enabled or not. To not enable the write zero feature (this is the default setting), the build option `OS_CACHE_WRITE_ZERO` must be set to a value of zero (0). To enable the feature, the build option `OS_CACHE_WRITE_ZERO` must be set to a non-zero.

The default setting is shown in the following table, and this section of code is located around line 70 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-28 OS_CACHE_WRITE_ZERO setting

```
  .if !($$defined(OS_CACHE_WRITE_ZERO))
OS_CACHE_WRITE_ZERO  .equ       0
  .endif
```

The default setting can be changed to enable the full line write of zero; this is shown in the next table:

Table 2-29 Enabling full line write of zero

```
  .if !($$defined(OS_CACHE_WRITE_ZERO))
OS_CACHE_WRITE_ZERO  .equ       1
  .endif
```

Alternatively, it is possible to overload the value assigned to `OS_CACHE_WRITE_ZERO` in `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting, as shown in the following example:

Table 2-30 Command line set of OS_CACHE_WRITE_ZERO

```
armcl  … -D OS_CACHE_WRITE_ZERO =1  …
```

NOTE: If the L1 cache is not enabled (`OS_SAME_L1_PAGE_TBL` is set to -1) or the L2 cache is not enabled (`OS_L2_BASE_ADDR` is zero), then the option `OS_CACHE_WRITE_ZERO` is internally considered as set to zero (the full line of write zero is not enabled).

The built option `OS_CACHE_WRITE_ZERO` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-9 GUI setting of OS_CACHE_WRITE_ZERO

2.2.10 L2C-310 Registers Base Address

The memory address where the L2C-310 Level 2 cache registers are located is not defined in reference to the processor peripheral base address. As such, each device uses its unique base address for the L2 cache registers. The build option `OS_PLATFORM` (see Section 2.2.3) and build option `OS_L2_BASE_ADDRESS` in the file `ARMv7_SMP_L1_L2_CCS.s` are used to specify the base address of the L2C-310 cache registers.

When the build option `OS_L2_BASE_ADDR` is set to the value of zero (0), the L2 cache is disabled. When it is set to minus one (-1), it means the base address of the L2C-310 registers is linker supplied, through the symbol `L2baseAddr`. If the build option `OS_L2_BASE_ADDR` value is neither zero (0) nor minus one (-1), then the value assigned to the build option is the base address of the L2C-310 cache registers. Finally, if the build option is NOT defined, the value is internally determined using the definition of the build option `OS_PLATFORM`.

The definition of the build option `OS_L2_BASE_ADDR` is located at a few places in the file `ARMv7_SMP_L1_L2_CCS.s`. The different declarations are inside a conditional block corresponding the definition of the build option `OS_PLATFORM`.

Table 2-31 L2 cache default base address

```
 .if !($$defined(OS_L2_BASE_ADDR))
OS_L2_BASE_ADDR .equ  0xFFFEF000
 .endif
```

It is possible to overload the `OS_L2_BASE_ADDR` value set in `ARMv7_SMP_L1_L2_CCS.s` by using the assembler (through the compiler) command line option `-D` and specifying the desired setting (here is to use the linker supplied symbol `L2baseAddr`) with the following:

Table 2-32 Command line set of `OS_L2_BASE_ADDR`

```
armcl  … -D OS_L2_BASE_ADDR=-1 …
```

The built option `OS_L2_BASE_ADDR` can also be set through CCS GUI in the *CCS Build* → *ARM Compiler* → *Advance Options* → *Predefined Symbols* menu as shown in the following figure:

Figure 2-10 GUI setting of `OS_L2_BASE_ADDR`

For example, the following platforms use these addresses:

Table 2-33 L2 base addresses

Target Platform	L2 base address
Altera / Cyclone V Soc FPGA	`0xFFFEF000`
Freescale / iMX6	`0x00A02000`
Texas Instruments / OMAP 4460	`0x48242000`
Xilinx / Zynq XC7Z020	`0xF8F02000`

2.2.11 Non-shared memory

The L1 cache and MMU allow each core to have non-shared (private) memory pages dedicated to them. The value specified by the build option `OS_USE_NON_SHARED` controls if private memory pages are set-up by the cache configuration module or not. To not support private memory (this is the default setting), the build option must be set to a value of zero (0). To set-up private memory areas (the areas are defined by the private memory definition table, see Section 2.3.3) the build option must be set to a value of non-zero.

The default setting, which is to not set-up private memory for the cores, a value of zero (0), is shown in the following table and is located around line 75 in the file `ARMv7_SMP_L1_L2_CCS.s`:

Table 2-34 `OS_USE_NON_SHARED` setting

```
  .if !($$defined(OS_USE_NON_SHARED))
OS_USE_NON_SHARED     .equ     0
  .endif
```

It is possible to overload the OS_USE_NON_SHARED value set in ARMv7_SMP_L1_L2_CCS.s by using the assembler (through the compiler) command line option -D and specifying the desired setting, as shown in the following example:

Table 2-35 Command line set of OS_USE_NON_SHARED

```
armcl  … -D OS_USE_NON_SHARED=1  …
```

The built option OS_USE_NON_SHARED can also be set through CCS GUI in the *CCS Build → ARM Compiler → Advance Options → Predefined Symbols* menu as shown in the following figure:

Figure 2-11 GUI setting of OS_USE_NON_SHARED

NOTE: Enabling the use of non-shared memory requires each core to have it own page table. An error message is issued during the assembly phase if the build option OS_USE_NON_SHARED is non-zero (to use non-shared memory) and the build option OS_SAME_L1_PAGE_TBL (see Section 2.2.4) is set to a non-zero value (all cores use the same page table).

2.2.12 ARM Cache Errata

ARM has issued a few patches to repair errata in the cache module. mAbassi's cache module has code that implements the known errata. By default, no errata code is inserted in the cache code. There are two ways to enable errata code. One is to set the build option OS_ARM_ERRATA_ALL to a non-zero value. Doing so will insert the code for all supported errata but, depending on the revision and variant of the core, only the applicable patches are applied. The other way is to only add specific erratum code. This is done by not defining OS_ARM_ERRATA_ALL (or by defining it and setting it to a value of zero) and then defining as non-zero values individual built options OS_ARM_ERRATA_*NNNNNN*, where *NNNNNN* is the ARM erratum number to activate. As for all build options, the setting of the cache errata build options can be performed either by changing the code directly in the file ARMv7_SMP_L1_L2_CCS.s or by a define on the assembler (through the compiler) command line (refer to previous sections on how to do this).

The following table shows all errata numbers for which the repair code was added in the file ARMv7_SMP_L1_L2_CCS.s. For further details on these errata, either go on ARM website, or perform a quick search on the Web.

Table 2-36 Cache Errata handle by ARMv7_SMP_L1_L2_CCS.s

Erratum	Revision	Description
742230	r1p0 to r2p2	DMB between 2 writes may not work.
742231	r2p0 to r2p2	In SMP, when 2 core access same line, data corruption could occur.
743622	All r2p*	Possible data corruption.
751472	All prior to r3p0	Interrupted ICIALLUIS may not complete (only needed in SMP mode).
753970	r3p0	Buffer can remain in the PL310 L2 cache after the sync operation is completed.
764369	All MPcore versions	Data cache line maintenance operation by MVA targeting an Inner Shareable memory region may fail. All code doing maintenance must add DSB instruction.

The following table shows ARM cache errata that cannot be corrected by the code in the file `ARMv7_SMP_L1_L2_CCS.s` as these errata require the application code to add perform special handling:

Table 2-37 Cache errata to be handle by the application

Erratum	Revision	Description
588369		Non applicable as the L2 cache supported is PL310 and not PL210.
720789	`All prior to r2p0`	Must invalidate all TLB entries when flushing.
754322	`All r2p* and r3p*`	A DSB instruction must be inserted before an ASID change.
754327	`All prior to r2p0`	Memory access barriers must be added in the application code.
775420	`r2p2 to r3p0`	See `Abort_Handler` in `mAbassi_SMP_CORTEXA9CCS.s`. If required, the data abort handler must be upgraded if recovery from the fault is needed.

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 (optional table)

At around line 140 in the file in `ARMv7_SMP_L1_L2_CCS.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 110, the real 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.

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.

The example shared memory definition table is located around line 85 in the file `ARMv7_SMP_L1_L2_CCS.s`, as shown in the following table:

Table 2-38 Example shared memory definition table

```
 .if ((OS_MMU_ALL_INVALID) != 0)
SharedInfo:                         ; Table to define the shared/cached memory areas
; ----------- SIZE ----- ADDRESS -- ; when the whole table sdefaults to invalid
   .long   0x00100000, 0x02000000    ; The table holds pairs of value (size, base) and
   .long   0                         ; is terminated with a size of 0
 .endif
```

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 220 in the file `ARMv7_SMP_L1_L2_CCS.s`. Taking the simple example described above, the table should be filled as follows:

Table 2-39 Example peripheral definition table

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

Although the Cache / MMU memory pages are 1 Mbyte (`0x00100000`), if the specified size for a block of peripheral registers is not an exact multiple of 0x00100000, it will be internally set at the `ceil` value to the next exact multiple of 1Mbyte. In the case of the base address, if the specified address for a block of registers is not an exact multiple of `0x00100000`, it will be internally set at the floored value to the next exact multiple of 1M. It is strongly advised to always specify the sizes and addresses in exact multiple of 1 Mbyte 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 `0x00001000` for a base address of `0x8FFFFF00`.

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`.

The table is referenced by the label `PrivateInfo`. 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 225 in the file `ARMv7_SMP_L1_L2_CCS.s`.

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

Table 2-40 Example private memory definition table

```
  .if ((OS_USE_NON_SHARED) != 0) & ((OS_N_CORE) > 1)
PrivateInfo:                          ; Table to define the core private memory area
; ------------ SIZE ------ V ADDR -- PH ADDR -- ; Terminated with 0 Size.
  .long  0x01000000, 0x80000000, 0x80000000     ; Triplets: Size / Virt addr / Phys addr
  .long  0                      ; Core #0 Non-shared: 0x80000000 mapped to 0x80000000

  .long  0x01000000, 0x80000000, 0x81000000
  .long  0                      ; Core #1 Non-shared: 0x80000000 mapped to 0x81000000

  .long  0x01000000, 0x80000000, 0x82000000
  .long  0                      ; Core #2 Non-shared: 0x80000000 mapped to 0x82000000

  .long  0x01000000, 0x80000000, 0x83000000
  .long  0                      ; Core #3 Non-shared: 0x80000000 mapped to 0x83000000

  .endif
```

NOTE: `COREcacheON()` does not check for the overlapping of physical memory pages between cores. If the definition table has such an error, the results are unpredictable.

3 Implementation

The cache configuration performs the following operations in this listed order

- Disabling of the L1 Caches & MMU & SCU.
- Invalidation and flushing of the L1 caches
- L1 cache is enabled
- The page table set-up for the whole memory as cached and shared
- The page table entries are set-up for the peripheral addresses (non-cached)
- The page table entries of all cores private memory is set to non-accessible
- The page table entries for the current core private memory are set up
- The MMU is set-up and enabled
- The SCU is set-up and enabled (Core #0 only)
- The L2 cache is set-up and enabled (Core #0 only)

Of the steps involved, only a few need explanation. These are the ones where the L1 page table is set-up. The explanation assume individual page table (the build option `OS_SAME_L1_PAGE_TBL` set to non-zero and not minus one (-1)).

The first step performed in filling the page table is to write in all 4096 entries of the table the configuration of the cache as coherent memory, and for the MMU to make all virtual addresses the same as the physical addresses. This means the whole addressing space is declared shared, meaning the caches are set be coherent for the whole address space.

The second step fills the entries related to the peripheral addresses held in the peripheral definition table. These entries are set as non-shared with caching disabled. There are no provisions to map the physical addresses of the peripheral addresses into different virtual ones.

The third and optional step (when `OS_USE_NON_SHARED` is non-zero) fills the entries in the page table that defines private (non-shared) memory pages. When a memory page is made private to a core, two things happen: the physical memory page is made unavailable to the other cores and the virtual address of the memory page is most likely different from the physical memory page. The first thing done when the information on the private memory page is inserted in the page table is to invalidate all the physical memory pages that are declared private, no matter if the definition is for the current core or another one. Doing so as a first step guarantees that all private pages are inaccessible. Then the private memory pages of the current core only are configured in the page table. It is this configuration that assigns the virtual page to the physical page and declares them as non-shareable and non-coherent.

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, the MMU and the SCU. 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/or 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 L1 and L2 caches lines are 32 bytes and because only full cache linescan be flushed, the real start address is the exact multiple by 32 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 32 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/or L2 cache
`Len:`	Number of consecutive addresses of the memory range to flush from the L1 and/or L2 cache.

Returns

`void`

Component type

Function

Options

Notes

See also

`DCacheInvalRange()`	Section 0

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/or 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 L1 and L2 caches lines are 32 bytes and because only full cache lines can be invalidated, the real start address is the exact multiple by 32 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 32 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/or L2 cache
`Len:`	Number of consecutive addresses of the memory range to invalidate from the L1 and/or 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 A9, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] Abassi – ARMv7 Caches