Freescale Semiconductor
Application Note

Document Number: AN3899
Rev. 1, 05/2009

Using the SC3000 Linker Control File for MSC8156 Applications

1 Introduction

This application note describes how to use the Linker Control File (LCF) to define the memory layout for an application executing on the MSC8156 board. The LCF is a text file created by the application developer and used by the linker to define the placement of data and code in memory for a given application. To accomplish this, the LCF needs to define the initial setup of the MSC8156 Memory Management Unit (MMU). The MMU offers a level of sophistication that may prove challenging for the first-time user. Therefore, this application note is provided to help you understand how to use the LCF to set up the MSC8156 MMU to define an application's memory map.

This document is concerned with the MSC8156 memory map as it is visible to an application executing on the SC3850 cores. From this point of view, you must consider two distinct memory maps when developing an MSC8156 application. These two maps are defined by the use of either physical addresses or virtual addresses. A physical address is the actual address of a device (memory or peripheral) within the MSC8156 DSP.

Contents

2 Overview

Directives in the LCF are used to define the memory for an MSC8156 application. The LCF consists of three files: `mmu_attr.l3k`, `common.l3k` and `msc8156.l3k`. The first two files are included in the third one. The subsequent sections focus on their description.

Please note that throughout the linker command files, the assert directives are used to halt the execution in case of wrong configuration. These directives are used to prevent the execution of the linker with bad input. Usually, these directives test that a specific value is in a certain range or is one of the values of an enumeration.

3 Output Section Definitions

This section focuses on the `msc8156.l3k` file that, among other things, groups input sections into output sections that are placed together in virtual memory areas. It is important to make a distinction between the output sections and descriptors in the MMU. The output sections are placed into virtual memories, and a descriptor is made up of:

- the start of the virtual memory,
- the start of the physical memory, and
- the size

Listing 1 shows how to locate private boot data in M2, M3, or DDR.

Listing 1. Locating private boot data in M2, M3, or DDR

```
descriptor__xxx__cacheable_wb__sys__private__data__boot
{
  LNK_SECTION (att_mmu, "rw", _MMU_TABLES_size, 0x4, ".att_mmu");
  LNK_SECTION (stack, "rw", _StackSize, 0x4,"stack");
  .ovltab
} > data_boot_c;
```

The above output section definition groups the `.ovltab`, `att_mmu`, and `stack` input sections. The `".att_mmu"` the data section is used in startup file in runtime library and system operation to set the MMU registers. The `.ovltab` data section is used by the overlay manager. The `LNK_SECTION` directive is explained in a later section. `data_boot_c` is a virtual memory area in which the output section is placed.

Listing 2 shows how private data is frequently accessed and placed in M3. It shows how an output section definition groups a longer list of input sections. The MMU settings are set later for the cacheable write back policy.

Listing 2. Accessing private data frequently and placing it in M3

```
descriptor__m3__cacheable_wb__sys__private__data
{
   ".m3__cacheable_wb__sys__private__data",
   "reserved_crt_tls",
   ".data",
   ".rom",
```

```
    ".m3__cacheable_wb__sys__private__rom",
    ".bsstab", ".init_table", ".rom_init",
    ".rom_init_tables", ".exception", ".exception_index", ".staticinit",
    ".m3__cacheable_wb__sys__private__bss",
    ".bss"
}> m3_private_data_c_wb;
```

The description of the parameters defined in Listing 2 are as follows:

- `reserved_crt_tls` — Data section that is used in the reentrant runtime library.
- `.bsstab` — Read-only data section that is used in the startup file to fill the `.bss` sections with zeros.
- `.init_table` — Read-only data section that is used to initialize the ROM global variable to RAM (`-mrom` option from `scc`).
- `.rom_init` — Compiler generated function, which must be placed in LCF if generated. It contains the initial values (constants) for non-constant variables.
- `.rom_init_tables` — Contains the `_rom_init_tables` symbol. It is used at startup for initializing the ROM variables to RAM. The difference between the `.init_table` and `.rom_init_tables` is that `.rom_init_tables` is generated by the linker. Same as `.init_table`, the information must be placed in a private descriptor.
- `.staticinit` — Read-only data section that is used in the startup file/runtime library to initialize the C++ static objects.
- `.rom` — Contains constants and is generated by compiler to be able to place constant data at a place different from writable data.

By means of the compiler application file, the parts of application data can be defined to end up in the `.m3__cacheable_wb__sys__private__data`, `.m3__cacheable_wb__sys__private__rom`, and `.m3__cacheable_wb__sys__private__bss` sections.

Listing 3 shows the output section for M3 non-cacheable shared data. A good example of an input section to be grouped here is the `reserved_crt_mutex` data section. This input section is used in the reentrant runtime library and contains the MUTEX variables defined in the `reserved_crt_mutex` data section. Given its synchronization purpose, the `reserved_crt_mutex` section has to be in a shared region such as M3 memory. In MSC8156 architecture, it has to be in a non-cacheable output section.

Listing 3. Output section for M3 non-cacheable shared data

```
descriptor__m3__non_cacheable_wt__sys__shared__data
{
    ".m3__non_cacheable_wt__sys__shared__data"
    "reserved_crt_mutex"
    ".m3__non_cacheable_wt__sys__shared__rom"
    ".m3__non_cacheable_wt__sys__shared__bss"
} > m3_shared_data_nc_wt;
```

Listing 4 and Listing 5 shows examples of virtual memory areas and the output sections that are placed in them.

Listing 4. Output section in private memory

```
unit private (*)
{
   MEMORY
   {
     data_boot_c ("rw") :
     org = _VIRTUAL_DATA_BOOT_start,
     len = _VIRTUAL_DATA_BOOT_size;
       // … other virtual memory entries
   }
   SECTIONS
   {
     descriptor__xxx__cacheable_wb__sys__private__data__boot
     {
        LNK_SECTION (att_mmu, "rw", _MMU_TABLES_size, 0x4, ".att_mmu");
        LNK_SECTION (stack, "rw", _StackSize, 0x4,"stack");
        .ovltab
     }
     // … other output sections
    }
}
```

Listing 5. Output section in shared memory

```
unit shared (*)
{
   MEMORY
   {
      m3_shared_textboot_c ("rx"): AFTER (m3_shared_data_c_wb);
       // … other virtual memory entries
   }

   SECTIONS
   {
     descriptor__m3__cacheable__sys__shared__text_boot
     {
       . = align (0x1000);  //restriction due to VBA register.
      _VBAddr =.;          // Virtual Base Address must be set at beginning of interrupt table
       .intvec
       .text_boot
     }> m3_shared_textboot_c;
       // … other output sections
    }
}
```

NOTE See Appendix C — Other Descriptor Settings for other examples of output sections.

4 Common Settings

The `common.l3k` file contains definitions, which are common to all cores.

4.1 Application Physical Memory Layout

The architecture is specified as a command line argument or using a directive in the LCF, as shown:

```
arch (msc8156);
```

After identifying the architecture, the linker defines the default values for the specified architecture, as shown below. These values include the size and borders of all physical memories:

```
_M2_size,_M2_start,_M2_end
_M3_size,_M3_start,_M3_end
_DDR_size,_DDR_start,_DDR_end
```

All the cores of the specified architecture are used unless you define the number of cores, using the following directive:

```
number_of_cores (number);
```

You can specify the value of the Status Register (SR) after reset, as shown:

```
_SR_Setting = 0x3e4000c;
```

You can specify the following settings by using the SR value:

- Exception mode
- Interrupt level 31
- Saturation mode enabled
- Rounding mode: nearest even

In the startup code, the first instruction, shown below, initializes the status register with the default settings.

```
move.l #_SR_Setting, sr
```

The `common.l3k` file defines another critical setting related to M2 memory configuration (referred to as M2/L2 cache settings in MSC8156).

Table 1shows the association of M2 memory configuration with _M2_Setting.

Table 1. Association of M2 memory configuration with _M2_Setting

M2 Size	_M2_Setting
0KB	0x00
64KB	0x01
128KB	0x03
192KB	0x07
256KB	0x0f

Table 1. Association of M2 memory configuration with _M2_Setting

M2 Size	_M2_Setting
320KB	0x1f
384KB	0x3f
448KB	0x7f
512KB	0xff

4.2 Application Virtual Memory Layout

There are two distinct virtual spaces; one for program and one for data (due to MMU implementation). These two spaces may overlap and have the range, 0...4G. You can define the size stack and heap, as follows:

```
_StackSize= 0x7f00;
_HeapSize= 0x1000;
```

The actual heap and stack are declared using the LNK_SECTION directives, as shown in Listing 6.

Listing 6. Declaring heap and stack

```
LNK_SECTION (stack,         //section type
             "rw",         //flags
             _StackSize, //length
             0x4,         //alignment
             "stack");  //name
LNK_SECTION (heap, "rw", _HeapSize, _HeapSize, "heap");
```

The LNK_SECTION directive defines an input section of type; stack, heap, att_mmu, or bss of the given size and alignment.

```
_StackStart= originof ("stack");
_TopOfStack= (endof ("stack") - 7) & 0xFFFFFFF8;
__BottomOfHeap= originof ("heap");
__TopOfHeap= (endof ("heap") - 7) & 0xFFFFFFF8;
```

where, originof and endof are intrinsics that return the address of the input section specified. The input section can be a regular input section (identified by its name) or a special input section (defined using the LNK_SECTION construct and identified by the name of the LNK_SECTION).

The private code is placed in M2 and its size is defined as:

```
_PRIVATE_M2_TEXT_size = 0x2000;
```

The size for private data to be placed in M2, M3, and DDR memory can be modified using the following symbols:

```
_PRIVATE_M2_DATA_size = _M2_size -
_PRIVATE_M2_TEXT_size - _DATA_BOOT_size;
```

```
_PRIVATE_M3_DATA_size= 0x10000;
_PRIVATE_DDR_DATA_size= 0x80000;
```

The boot descriptor in the `common.l3k` file is placed in M2, M3, or DDR memory, and contains the stack and MMU table. The boot descriptor size must be a power of 2 and it is represented by the following expression.

```
_DATA_BOOT_size = _MMU_TABLES_size + _StackSize;
```

NOTE The boot descriptor must be the first descriptor in the Memory Attributes and Translation Table (MATT).

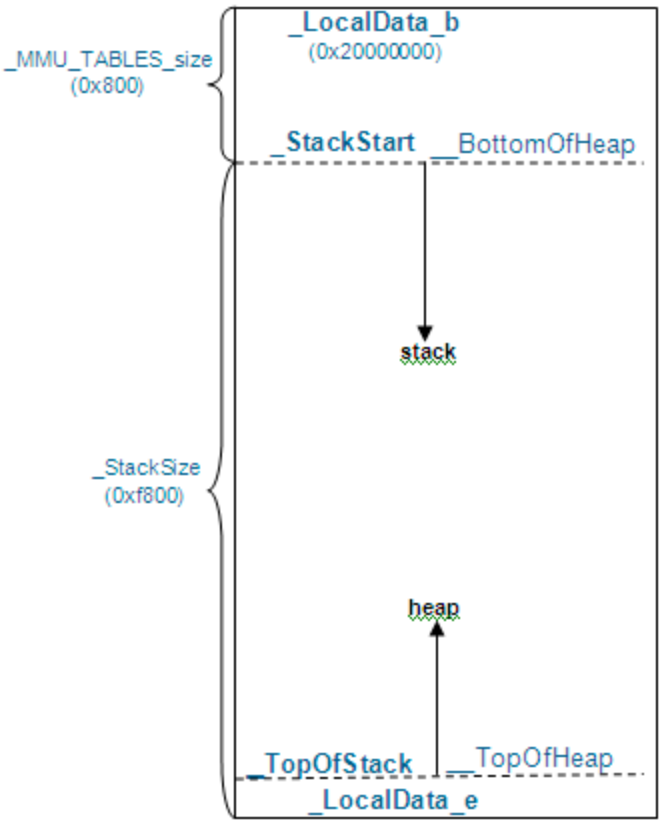
By default, the boot descriptor is placed in the output section as:

```
descriptor__xxx__cacheable_wb__sys__private__data__boot
```

Figure 1. shows the virtual memory view of the boot descriptor in dynamic stack-heap configuration, where:

```
_LocalData_b = _VIRTUAL_DATA_BOOT_start;
_LocalData_size = _DATA_BOOT_size;
_LocalData_e = _LocalData_b +_LocalData_size - 1;
```

Figure 1. Virtual Memory View of Boot Descriptor in Dynamic Stack-Heap Configuration

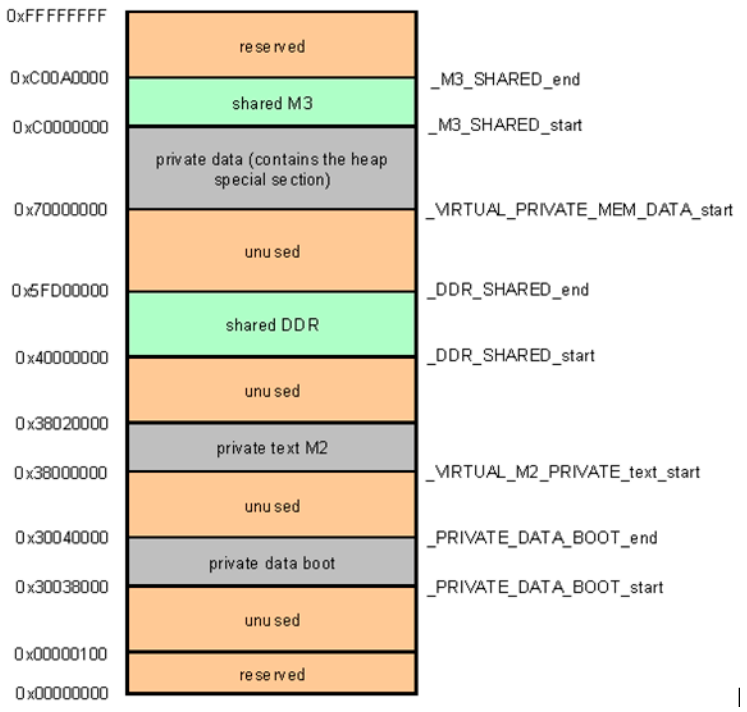


The virtual memory map placement is done according to the following definitions:

```
_VIRTUAL_PRIVATE_MEM_DATA_start= 0x70000000;
_VIRTUAL_DATA_BOOT_start= 0x20000000;
_VIRTUAL_M2_PRIVATE_text_start= 0x38000000;
```

Figure 2. shows a visual representation of the virtual memory for a sample configuration.

Figure 2. Virtual Memory Layout



In physical space, code and data are mapped to the specific memories, for example:

- shared IPC data is mapped in M3
- private data is mapped to M2, M3, and DDR
- shared data and code is mapped to M3 and DDR
- private code stays in M2

The descriptor for `att_mmu` tables and stack is placed at the end of M2, as shown:

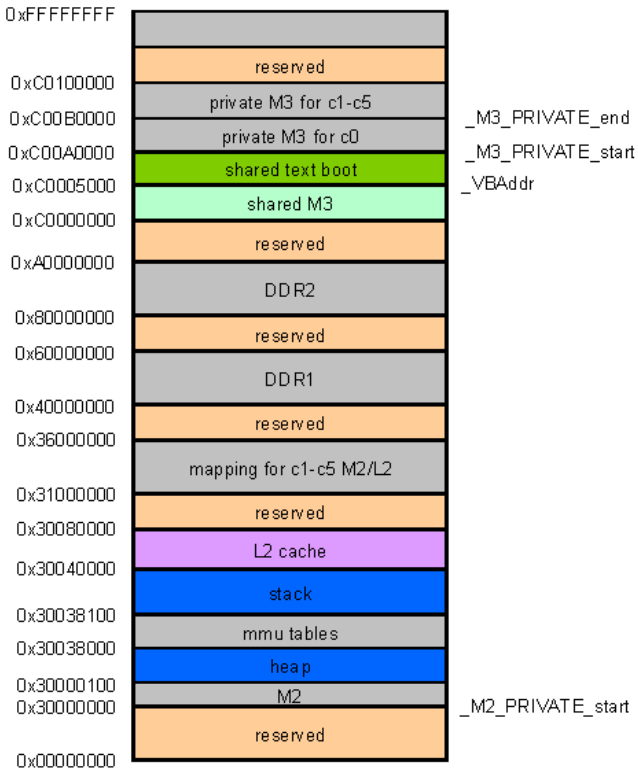
```
_PRIVATE_DATA_BOOT_start =_M2_end - _DATA_BOOT_size + 1;
_PRIVATE_DATA_BOOT_end =_PRIVATE_DATA_BOOT_start +
_LocalData_size;
```

The startup code uses some special symbols, such as `_LocalData_Phys_b`, to create the first descriptor. The following directive defines `_LocalData_Phys_b`:

```
_LocalData_Phys_b = _PRIVATE_DATA_BOOT_start-(core_id
()*0x01000000);
```

Figure 3. shows a visual representation of the physical memory for a sample configuration.

Figure 3. Physical Memory Layout



5 Virtual to Physical Space Mappings

The `address_translation` directive is used to map virtual memories to physical memories.

Listing 7 shows an example of the `address_translation` directive for a private data boot descriptor.

Listing 7. The `address_translation` directive for private data boot descriptor

```
address_translation (*)
{
  // … other address_translation entries
  data_boot_c (SYSTEM_DATA_MMU_DEF): M2,org = _PRIVATE_DATA_BOOT_start;
}
```

The `address_translation` directive maps the `data_boot_c` virtual memory to the M2 physical memory, starting at the address in `_PRIVATE_DATA_BOOT_start`. The attributes are also specified by an expression enclosed in parentheses. All the information about the virtual memory is extracted from its definition in the `MEMORY` section of the unit directive.

The descriptor__m3__cacheable__sys__shared__text_boot descriptor need to be mapped 1:1 (physical and virtual shared the same value), because the boot code and interrupt vector are put in this descriptor. The following listing shows another example of the `address_translation` directive.

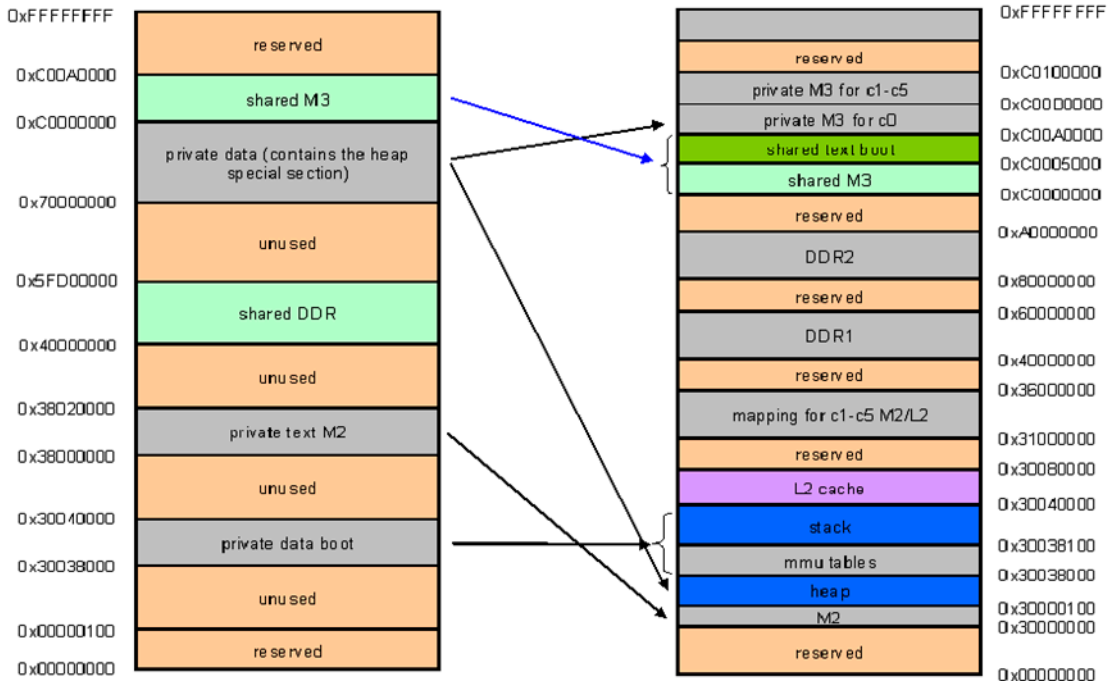
```
address_translation (*) map11
{    // ... other address_translation entries
   m3_shared_textboot_c (SYSTEM_PROG_MMU_DEF): M3;
}
```

Because the first 12 least significant bits in the VBA Reset Value register (VBA_RST_VAL) are reserved and must have the value zero, the first directive in the descriptor must be

```
. = align (0x1000);
```

The map11 keyword placed in the address_translation directive applies to all address translation entries and enforces one to one mapping. Figure 4. shows the mapping between virtual and physical memory.

Figure 4. Mapping Between Virtual and Physical Memory



6 MMU and Cache Configuration

The mmu_attr.13k file provides symbol definitions for MMU configuration and cache enablement. The following naming conventions are used for defining symbols in the mmu_aatr.13k file for individual settings in MMU descriptors:

- the symbol name starts with MMU, followed by PROG for program/text descriptors and DATA for data descriptors
- the symbol name is suffixed with one of the following:
  - cache attribute, for example, CACHEABLE, L2CACHEABLE
  - burst size, for example, BURST_SIZE_4, BURST_SIZE_2

— access permissions, such as:
  – read(RPERM), write(WPERM) or execute(XPERM)
  – USER or SUPER
  – SHARED, for example, RPERM_USER, WPERM_SUPER, SHARED

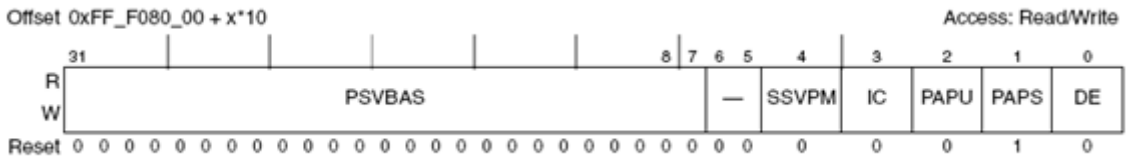
Listing 8 shows a sample of M_PSDAx (Program Segment Descriptor Registers A) symbol definitions in the mmu_attr.l3k file.

Listing 8. M_PSDAx sample symbol definitions

```
MMU_PROG_DEF_SHARED=0x00000010;// SSVPM bit [4]
MMU_PROG_DEF_CACHEABLE=0x00000008;// IC bit [3]
MMU_PROG_DEF_XPERM_USER=0x00000004;// PAPU bit [2]
MMU_PROG_DEF_XPERM_SUPER=0x00000002;// PAPS bit [1]
```

Figure 5. shows how the M_PSDAx register is stored in memory.

Figure 5. M_PSDAx Register



The mmu_attr.l3k file contains the definitions for other parameters as well, such as default attributes for user program descriptors. Listing 9 shows an example.

Listing 9. Default attributes for user program descriptors

```
USER_PROG_MMU_DEF=MMU_PROG_DEF_CACHEABLE     |
                  MMU_PROG_PREFETCH_ENABLE   |
                  MMU_PROG_L2CACHEABLE       |
                  MMU_PROG_NEXT_LINE_PFETCH  |
                  MMU_PROG_DEF_XPERM_USER    |
                  MMU_PROG_DEF_XPERM_SUPER   |
                  MMU_PROG_BURST_SIZE_4
```

NOTE The rest of the symbols defined for MMU attributes are described in Appendix B — MMU Attributes.

You can specify protection, translation and alignment for MMU and cache by modifying the default symbol definitions that Listing 10 shows.

Listing 10. Default symbol definitions for protection, translation, and alignment for MMU and cache

```
_ENABLE_MMU_PROTECTION =1;// MPE [3]
   // -1 = the MMU memory protection is off

_ENABLE_MMU_TRANSLATION =1;// ATE [2]
```

```
  // -1 = the MMU translation is off

_ENABLE_MMU_DATA_NON_ALIGNED =1; // DNAMEE [6]
       // -1 = Data Non-aligned Memory Exception is off

_ENABLE_CACHE =1;
     // -1 = all types of cache are not enabled. Enables L2_CR2 [CE], IC_CR2 [CE], DC_CR2 [CE]
```

Similarly, you can modify the _ENABLE_VTB symbol for Virtual Trace Buffer (VTB) reservation, such that:

- when the value is set to 1, the VTB gets reserved in M2 memory
- when the value is set to 2, the VTB gets reserved in M3 memory
- for any other value, the VTB is not configured automatically

7 C++ Support

Add the source code of Listing 11 to the LCF to add C++ support.

Listing 11. Adding C++ support to the LCF

```
// By default exception support is enabled
// This value can be overwritten by linker cmd line options
ENABLE_EXCEPTION = 0x1;

// Define the static initializer section required for C++ startup
_cpp_staticinit_start = originof (".staticinit");
_cpp_staticinit_end = _cpp_staticinit_start + sizeof (".staticinit");

__exception_table_start__ = ENABLE_EXCEPTION ? originof (".exception_index") : 0;
__exception_table_end__ = ENABLE_EXCEPTION ? (__exception_table_start__+
sizeof(".exception_index")) : 0;
```

8 Setting VTB

VTB can be reserved in M2 or M3 memory. The actual placement is done according to the _ENABLE_VTB symbol, as shown in Table 2.

Table 2. VTB Reservation in Physical Memory

_ENABLE_VTB	Physical Memory
0	M2
1	M3
2	-

Listing 12 shows how to reserve memory for VTB.

Listing 12. Reserving VTM in M2 and M3 memory

```
//set VTB start address and size for M2 and M3
_M2_VTB_size = 0x800;// 4K for each core
_M3_VTB_size = 0x8000;// 32K for each core

_VTB_size = (_ENABLE_VTB == 1) ? _M2_VTB_size : (_ENABLE_VTB == 2) ? _M3_VTB_size : 0x0;

_VTB_start = (_ENABLE_VTB == 1) ? _PRIVATE_DATA_BOOT_start - _M2_VTB_size : (_ENABLE_VTB == 2)
? _M3_PRIVATE_end - _M3_VTB_size + 1 : 0x0;

physical_memory private (*)
{
   reserve : org = _VTB_start, len = _VTB_size;
}
```

9 Moving Heap from M2 to M3

To move the `heap` section from M2 to M3, the `LNK_SECTION` directive that places the `heap` in a virtual memory (which is mapped to M2) must be moved to a descriptor placed into this virtual memory. Listing 13 shows how to move the heap from M2 to M3.

Listing 13. Moving heap from M2 to M3

```
descriptor__m2__cacheable_wb__sys__private__data
{
  .zdata
  .m2__cacheable_wb__sys__private__rom
  .m2__cacheable_wb__sys__private__data
  .m2__cacheable_wb__sys__private__bss
  LNK_SECTION (heap, "rw", _HeapSize, _HeapSize, "heap");
}> m2_private_data_c_wb;

descriptor__m3__cacheable_wb__sys__private__data
{
  .m3__cacheable_wb__sys__private__data
  reserved_crt_tls
  .data
  .m3__cacheable_wb__sys__private__rom
  .bsstab
  .init_table
  .rom_init
  .rom_init_tables
  .exception
  .exception_index
  .staticinit
  .m3__cacheable_wb__sys__private__bss
  .bss
  // place LNK_SECTION anywhere in this descriptor
}> m3_private_data_c_wb;
```

NOTE It is recommended to place `heap`, `stack`, and `bss` sections at the end of an output section. Otherwise, the output file grows in size because these sections require space, which is reserved in the output file using padding.

10 Moving Stack from M2 to M3

The `stack` and `att_mmu` sections are placed in the `data_boot_c` virtual memory. The moving of the `stack` section implies mapping this virtual memory area to M3 memory, as shown in the following listing. The `stack` and `att_mmu` sections must be placed together so that they can be moved together.

```
address_translation (*)
{
data_boot_c (SYSTEM_DATA_MMU_DEF) : M3, org =
_PRIVATE_DATA_BOOT_start;
// ... other address translation entries
}
```

In addition, you need to update the symbols defined in Listing 14. In Listing 14, the `stack` and `att_mmu` sections are placed at the end of the shared part of M3 memory.

Listing 14. Updating symbols to move `stack` from M2 to M3

```
_PRIVATE_DATA_BOOT_start = _M2_end - _DATA_BOOT_size + 1;
_LocalData_Phys_b = _PRIVATE_DATA_BOOT_start - (core_id () * 0x01000000);
_PRIVATE_DATA_BOOT_start = _M3_SHARED_end - (_NUMBER_OF_CORES * _DATA_BOOT_size) +
core_id () * _DATA_BOOT_size + 1;
_LocalData_Phys_b = _PRIVATE_DATA_BOOT_start - (core_id () * _DATA_BOOT_size);
```

M3 is not an on-core memory like M2; therefore, `_PRIVATE_DATA_BOOT_start` must be different for each core and must depend on the `core_id ()` intrinsic.

11 Appendix A — DDR Settings

11.1 Defining Physical Memory for DDR Private Data

The private space is placed at the beginning of the DDR memory if the size of private space is bigger than the size of shared space (see Listing 15).

Listing 15. Defining physical memory for DDR private data

```
_DDR_PRIVATE_start =
                     (_PRIVATE_DDR_DATA_size < _DDR_size -
                     (_NUMBER_OF_CORES * _PRIVATE_DDR_DATA_size)) ?
                     _DDR_start + _DDR_size -
                     (_NUMBER_OF_CORES * _PRIVATE_DDR_DATA_size) +
                     (core_id() * _PRIVATE_DDR_DATA_size) :
                     _DDR_start + (core_id() * _PRIVATE_DDR_DATA_size);
```

11.2 Defining Physical Memory for DDR Shared Data and Code

The shared space is placed at the beginning of DDR if the size of shared space is bigger than the size of private space (see Listing 16).

Listing 16. Defining physical memory for DDR shared data and code

```
_DDR_SHARED_start =
                    (_PRIVATE_DDR_DATA_size < _DDR_size -
                    (_NUMBER_OF_CORES * _PRIVATE_DDR_DATA_size)) ?
                     _DDR_start :
DDR_start + (_NUMBER_OF_CORES * _PRIVATE_DDR_DATA_size);
```

12 Appendix B — MMU Attributes

12.1 M_PSDBx (Program Segment Descriptor Registers B)

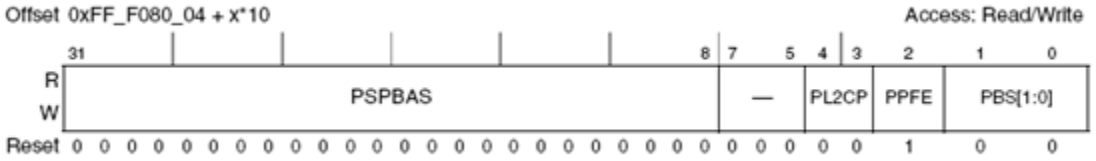
Listing 17 shows a sample of M_PSDBx register's symbol definitions.

Listing 17. M_PSDBx register's symbol definitions

```
MMU_PROG_L2CACHEABLE =0x00080000; // PL2CP bit [3, 4]
MMU_PROG_PREFETCH_ENABLE = 0x00040000; // PPFE bit [2]
MMU_PROG_BURST_SIZE_4 =0x00020000; // PBS bit [1, 0] VBR =4
MMU_PROG_BURST_SIZE_2 =0x00010000; // PBS bit [1, 0] VBR =2
MMU_PROG_BURST_SIZE_1 =0x00000000; // PBS bit [1, 0] VBR =1
```

Figure 6. shows how M_PSDBx is reserved in memory.

Figure 6. M_PSDBx Register



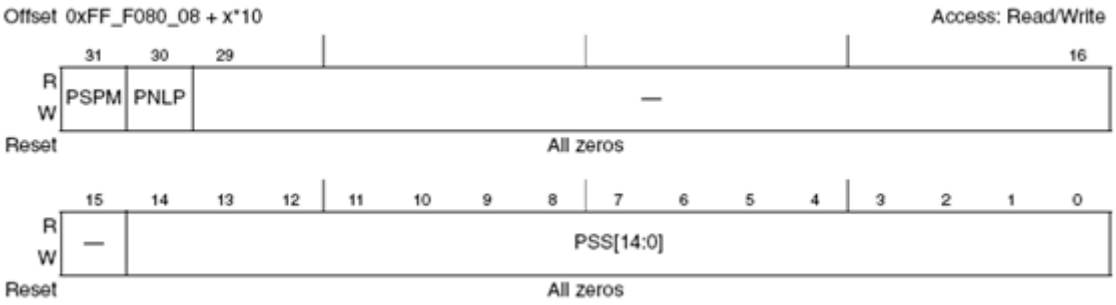
12.2 M_PSDCx (Program Segment Descriptor Registers C)

The symbol definitions for M_PSDCx are defined as:

```
MMU_PROG_NEXT_LINE_PFETCH =0x00004000; //PNLP bit [30]
```

Figure 7. shows how M_PSDCx is reserved in memory.

Figure 7. M_PSDCx Register



12.3 M_DSDAx (Data Segment Descriptor Registers A)

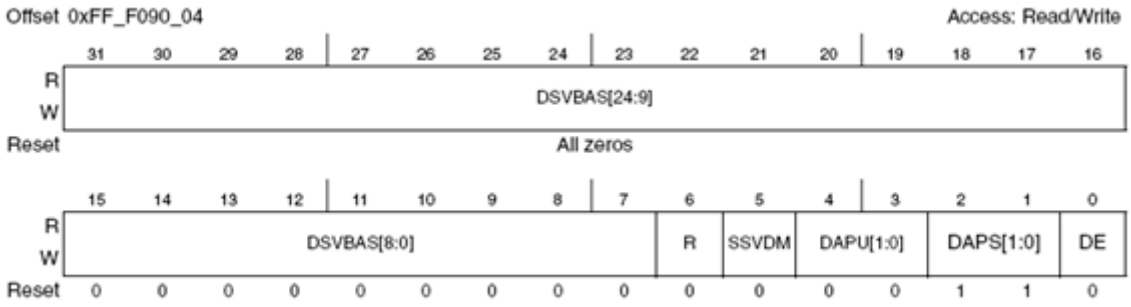
Listing 18 shows a sample of M_DSDAx register's symbol definitions.

Listing 18. M_DSDAx register's symbol definitions

```
MMU_DATA_DEF_SHARED=0x00000020; // SSVDM [5]
MMU_DATA_DEF_RPERM_USER=0x00000010; // DAPU [4, 3]
MMU_DATA_DEF_WPERM_USER=0x00000008; // DAPU [4, 3]
MMU_DATA_DEF_RWPERM_USER=0x00000018; // DAPU [4, 3]
MMU_DATA_DEF_RPERM_SUPER=0x00000004; // DAPS [2, 1]
MMU_DATA_DEF_WPERM_SUPER=0x00000002; // DAPS [2, 1]
```

Figure 8. shows how M_DSDAx is reserved in memory.

Figure 8. M_DSDAx Register



12.4 M_DSDBx (Data Segment Descriptor Registers B)

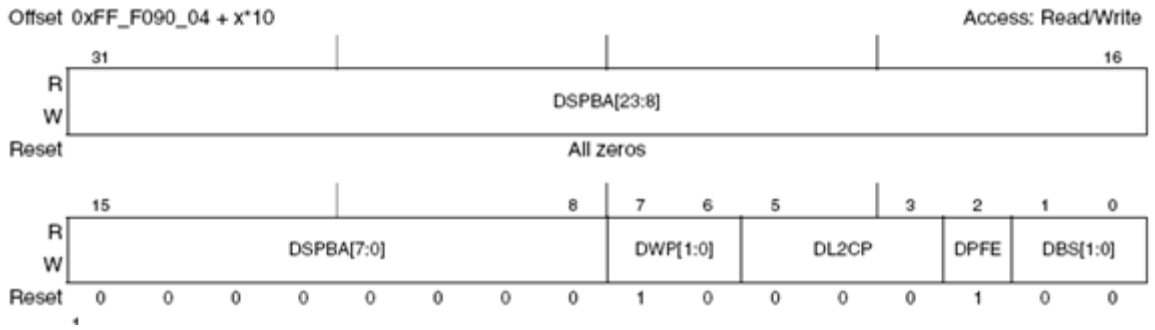
Listing 19 shows a sample of M_DSDBx register's symbol definitions.

Listing 19. M_DSDBx register's symbol definitions

```
MMU_DATA_NONCACHEABLE_WRITE_THROUGH = 0x00800000; // DWP [6, 7]
MMU_DATA_CACHEABLE_WRITE_BACK = 0x00400000; // DWP [6, 7]
MMU_DATA_CACHEABLE_WRITE_THROUGH = 0x00000000; // DWP [6, 7]
MMU_DATA_L2CACHEABLE_WRITE_THROUGH = 0x00000000; // DL2CP [5, 3]
MMU_DATA_L2CACHEABLE_ADAPTIVE_WRITE = 0x00180000; // DL2CP [5, 3]
MMU_DATA_L2NONCACHEABLE= 0x00100000; // DL2CP [5, 3]
MMU_DATA_L2CACHEABLE_WRITE_BACK = 0x00080000; // DL2CP [5, 3]
MMU_DATA_PREFETCH_ENABLE =0x00040000; // DPFE [2]
MMU_DATA_BURST_SIZE_4=0x00020000; // DBS [1, 0]
MMU_DATA_BURST_SIZE_2=0x00010000; // DBS [1, 0]
MMU_DATA_BURST_SIZE_1=0x00000000; // DBS [1, 0]
```

Figure 9. shows how M_DSDBx is reserved in memory.

Figure 9. `M_DSDBx` Register



12.5 Default Attributes for Shared Program Descriptors

Listing 20 shows how to set the default attributes for shared program descriptors.

Listing 20. Default attributes for shared program descriptors

```
SHARED_PROG_MMU_DEF =MMU_PROG_DEF_CACHEABLE|
                     MMU_PROG_PREFETCH_ENABLE|
                     MMU_PROG_L2CACHEABLE|
                     MMU_PROG_NEXT_LINE_PFETCH|
                     MMU_PROG_DEF_SHARED|
                     MMU_PROG_DEF_XPERM_USER|
                     MMU_PROG_DEF_XPERM_SUPER|
                     MMU_PROG_BURST_SIZE_4;
```

12.6 Default Attributes for System Program Descriptors

To set the default attributes for system program descriptors, set:

```
SYSTEM_PROG_MMU_DEF=SHARED_PROG_MMU_DEF;
```

12.7 Cacheable Private Settings

Listing 21 shows how to define the cacheable private settings.

Listing 21. Cacheable private settings

```
USER_DATA_MMU_DEF= MMU_DATA_CACHEABLE_WRITE_BACK|
                   MMU_DATA_PREFETCH_ENABLE|
                   MMU_DATA_L2CACHEABLE_WRITE_BACK|
                   MMU_DATA_DEF_RWPERM_USER|
                   MMU_DATA_DEF_RWPERM_SUPER|
MMU_DATA_BURST_SIZE_4;
```

12.8 Non-Cacheable Shared Data Settings

Listing 22 shows how to define the non-cacheable shared data settings.

Listing 22. Non-cacheable shared data settings

```
SHARED_DATA_MMU_DEF= MMU_DATA_NONCACHEABLE_WRITE_THROUGH|
                     MMU_DATA_L2NONCACHEABLE|
                     MMU_DATA_PREFETCH_ENABLE|
                     MMU_DATA_DEF_SHARED|
                     MMU_DATA_DEF_RWPERM_USER|
                     MMU_DATA_DEF_RWPERM_SUPER|
                     MMU_DATA_BURST_SIZE_4;
```

12.9 Cacheable Shared Settings

Listing 23 shows how to define the cacheable shared settings.

Listing 23. Cacheable shared settings

```
SYSTEM_DATA_MMU_DEF =MMU_DATA_CACHEABLE_WRITE_BACK|
                     MMU_DATA_PREFETCH_ENABLE|
                     MMU_DATA_L2CACHEABLE_WRITE_BACK|
                     MMU_DATA_DEF_SHARED|
                     MMU_DATA_DEF_RWPERM_USER|
                     MMU_DATA_DEF_RWPERM_SUPER|
                     MMU_DATA_BURST_SIZE_4;
```

13 Appendix C — Other Descriptor Settings

13.1 Shared Data in M3

The descriptor properties for shared data in M3 memory are:

- cacheable write back
- prefetch is enabled
- read and write access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 24 shows the descriptor settings defined for shared data in M3.

Listing 24. Descriptor settings for shared data in M3

```
descriptor__m3__cacheable_wb__sys__shared__data
{
   .m3__cacheable_wb__sys__shared__data
   .m3__cacheable_wb__sys__shared__rom
   .m3__cacheable_wb__sys__shared__bss
} > m3_shared_data_c_wb;
```

13.2 Private Text in M2

The descriptor properties for private text in M2 memory are:

- cacheable
- prefetch is enabled
- execute access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 25 shows the descriptor settings defined for private text in M2.

Listing 25. Descriptor settings for private text in M2

```
descriptor__m2__cacheable__sys__private__text
{
  .m2__cacheable__sys__private__text
} > m2_private_text_c;
```

13.3 Shared Text in M3

The descriptor properties for shared text in M3 memory are:

- cacheable
- prefetch is enabled

- execute access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 26 shows the descriptor settings defined for shared text in M3.

Listing 26. Descriptor settings for shared text in M3

```
descriptor__m3__cacheable__sys__shared__text
{
  .m3__cacheable__sys__shared__text
  .text
  .default
} > m3_shared_text_c;
```

13.4 Private Data in DDR

The descriptor properties for private data in DDR memory are:

- cacheable write back
- prefetch is enabled
- read and write access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 27 shows the descriptor settings defined for private data in DDR.

Listing 27. Descriptor settings for private data in DDR

```
descriptor__ddr__cacheable_wb__sys__private__data
{
  .ddr__cacheable_wb__sys__private__data
  .ddr__cacheable_wb__sys__private__rom
  .ddr__cacheable_wb__sys__private__bss
} > ddr_private_data_c_wb;
```

13.5 Non-Cacheable Shared Data in DDR

The descriptor properties for non-cacheable shared data in DDR memory are:

- non-cacheable write through
- prefetch is enabled
- read and write access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 28 shows the descriptor settings defined for non-cacheable shared data in DDR.

Listing 28. Descriptor settings for non-cacheable shared data in DDR

```
descriptor__ddr__non_cacheable_wt__sys__shared__data
{
  .ddr__non_cacheable_wt__sys__shared__data
  .ddr__non_cacheable_wt__sys__shared__rom
  .ddr__non_cacheable_wt__sys__shared__bss
} > ddr_shared_data_nc_wt;
```

13.6 Cacheable Shared Data in DDR

The descriptor properties for cacheable shared data in DDR memory are:

- cacheable write back
- prefetch is enabled
- read and write access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 29 shows the descriptor settings defined for cacheable shared data in DDR.

Listing 29. Descriptor settings for cacheable shared data in DDR

```
descriptor__ddr__cacheable_wb__sys__shared__data
{
  .ddr__cacheable_wb__sys__shared__data
  .ddr__cacheable_wb__sys__shared__rom
  .ddr__cacheable_wb__sys__shared__bss
} > ddr_shared_data_c_wb;
```

13.7 Shared Text in DDR

The descriptor properties for shared text in DDR memory are:

- cacheable
- prefetch is enabled
- execute access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 30 shows the descriptor settings defined for shared text in DDR.

Listing 30. Descriptor settings for shared text in DDR

```
descriptor__ddr__cacheable__sys__shared__text
{
  .ddr__cacheable__sys__shared__text
  .unlikely
} > ddr_shared_text_c;
```

13.8 Private Data in M2

The descriptor properties for private data in M3 memory are:

- cacheable write back
- prefetch is enabled
- read and write access in both user and supervisor mode
- burst size 4
- system task (shared between tasks)

Listing 31 shows the descriptor settings defined for private data in M2.

Listing 31. Descriptor settings for private data in M2

```
descriptor__m2__cacheable_wb__sys__private__data
{
  .zdata
  .m2__cacheable_wb__sys__private__rom
  .m2__cacheable_wb__sys__private__data
  .m2__cacheable_wb__sys__private__bss
} > m2_private_data_c_wb;
```

NOTE The `.zdata` section may be generated while using `-Xllt -zdata1`. Otherwise it is zero.

14 Appendix D — MMU Predefines 8156

The MMU predefines shown in Listing 32 are used for the MSC8156 architecture.

Listing 32. MMU predefines for the MSC8156 architecture

```
// temporary LCF for msc8156
_M2_Setting = 0x0f;
_M2_size =(_M2_Setting == 0x01) ? 0x10000:
          (_M2_Setting == 0x03) ? 0x20000:
          (_M2_Setting == 0x07) ? 0x30000:
          (_M2_Setting == 0x0f) ? 0x40000:
          (_M2_Setting == 0x1f) ? 0x50000:
          (_M2_Setting == 0x3f) ? 0x60000:
          (_M2_Setting == 0x7f) ? 0x70000:
          (_M2_Setting == 0xff) ? 0x80000:
          0x0;                                // M2 size.

physical_memory shared (*)
{
    M3:  org = _M3_start, len = _M3_size;
    DDR: org = _DDR_start, len = _DDR_size;
    DDR2: org = _DDR2_start, len = _DDR2_size;
}

physical_memory private (*)
{
    M2: org = _M2_start, len = _M2_size;
}

_M2_start = 0x30000000 + 0x1000000 * core_id();
_M2_end = _M2_start + _M2_size - 1;
_M3_start = 0xC0000000;
_M3_size = 0x00100000; // M3 size. (1M)
_M3_end = _M3_start + _M3_size - 1;
_DDR_start = 0x40000000;
_DDR_size = 0x20000000; // DDR size (512M)
_DDR_end = _DDR_start + _DDR_size -1;
_DDR2_start = 0x80000000;
_DDR2_size = 0x20000000;// DDR2 size (512M)
_DDR2_end = _DDR2_start + _DDR2_size -1;

// MMU ATTRIBUTES
MMU_PROG_L2CACHEABLE=0x00080000;
MMU_PROG_PREFETCH_ENABLE=0x00040000;
MMU_PROG_BURST_SIZE_4=0x00020000;
MMU_PROG_BURST_SIZE_1=0x00000000;
MMU_PROG_DEF_SHARED=0x00000010;
MMU_PROG_DEF_CACHEABLE=0x00000008;
MMU_PROG_DEF_XPERM_USER=0x00000004;
MMU_PROG_DEF_XPERM_SUPER=0x00000002;
MMU_PROG_DEF_SYSTEM=MMU_PROG_DEF_SHARED;
_MMU_PROG_DEF_SYSTEM=MMU_PROG_DEF_SHARED;
MMU_PROG_NEXT_LINE_PFETCH=0x00004000;

MMU_DATA_NONCACHEABLE_WRITE_THROUGH=0x00800000;
MMU_DATA_CACHEABLE_WRITE_BACK=0x00400000;
MMU_DATA_CACHEABLE_WRITE_THROUGH=0x00000000;
MMU_DATA_L2CACHEABLE_WRITE_THROUGH=0x00000000;
MMU_DATA_L2CACHEABLE_ADAPTIVE_WRITE=0x00180000;
MMU_DATA_L2NONCACHEABLE=0x00100000;
MMU_DATA_L2CACHEABLE_WRITE_BACK=0x00080000;
```

```
MMU_DATA_PREFETCH_ENABLE=0x00040000;
MMU_DATA_BURST_SIZE_4=0x00020000;
MMU_DATA_BURST_SIZE_1=0x00000000;

MMU_DATA_DEF_SHARED=0x00000020;
MMU_DATA_DEF_RPERM_USER=0x00000010;
MMU_DATA_DEF_WPERM_USER=0x00000008;
MMU_DATA_DEF_RPERM_SUPER=0x00000004;
MMU_DATA_DEF_WPERM_SUPER=0x00000002;
MMU_DATA_DEF_SYSTEM=MMU_DATA_DEF_SHARED;
_MMU_DATA_DEF_SYSTEM=MMU_DATA_DEF_SHARED;
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3899

June 1, 2009