Freescale Semiconductor
Application Note

Document Number: AN4497

CodeWarrior Linker Command File (LCF) for Qorivva/PX

1 Introduction

This document provides the steps to create LCF from scratch and explains common as well as unique application requirements handled in LCF using examples.

For detailed information on the Qorivva/PX Architectures LCF refer to the CodeWarrior Development Studio for Power Architecture® Processors Build Tools Reference Manual. You can find this document in `{MCU10.xinstallation path}\MCU\Help\PDF\MCU_Power-Architecture_Compiler.pdf`

2 Preliminary Background

The LCF along with other compiler directives, places pieces of code and data into ROM and RAM. You can do this by creating specific sections in the LCF and then matching them to the source code using pragma directives.

LCF consists of three kinds of segments, which must be in this order:

Contents

- A memory segment, which begins with the `MEMORY{}` directive,
- An optional closure segment, which begins with the `FORCE_ACTIVE{}`, `KEEP_SECTION{}`, or `REF_INCLUDE{}` directives, and
- A sections segment, which begins with the `SECTIONS{}` directive.

3 Creating an LCF from Scratch

Consider creating a LCF for a sample `BOOKE ROM` project.

First add the memory area for vectors (interrupts), exception tables, code, data, heap and stack.

Listing 1. Adding memory area

```
MEMORY
{
    resetvector:           org = 0x00000000,   len = 0x00000008
    exception_handlers_p0: org = 0x00001000,   len = 0x00001000
    internal_flash:        org = 0x00003000,   len = 0x001FD000

    internal_ram:          org = 0x40000000,   len = 0x0007C000
    heap  :                org = 0x4007C000,   len = 0x00002000 /*  Heap start location */
    stack :               org = 0x4007E000,   len = 0x00002000 /* Start location for Stack */
}
```

Place the sections to the above memory areas in LCF in the SECTIONS { } block.

```
.__bam_bootarea LOAD (0x00000000): {} > resetvector
```

The code to handle exceptions are grouped and placed in memory area `exception_handlers_p0`.

Listing 2. Grouping exceptions code

```
    GROUP : {
      .ivor_branch_table_p0  LOAD (0x00001000)   : {}
      .intc_hw_branch_table_p0 LOAD (0x00001800): {}
      .__exception_handlers_p0   LOAD (0x00001100) : {}
    } > exception_handlers_p0
```

The hardware initialization routines, application code, constants, code for constructors/destructors, and C++ exception tables are grouped together and placed in Flash.

Listing 3. Grouping initialization routines

```
   GROUP : {
      .intc_sw_isr_vector_table_p0 ALIGN (2048) : {}

      .init : {}
      .text : {}
       .rodata (CONST) : {
         *(.rdata)
         *(.rodata)
       }
       .ctors : {}
       .dtors : {}
       extab : {}
```

```
    extabindex : {}
  } > internal_flash
```

The uninitialized and initialized data are placed in RAM.

Listing 4. Placing data in RAM

```
GROUP : {
    .__uninitialized_intc_handlertable ALIGN(2048) : {}
    .data    : {}
    .sdata   : {}
    .sbss    : {}
    .sdata2  : {}
    .sbss2   : {}
    .bss     : {}
 } > internal_ram
```

NOTE For more information on placing data in RAM refer to `MCU_Power_Architecture_Compiler.pdf`.

The sections are allocated to segments in the order given in `SECTIONS/GROUP` block of lcf file.

For `internal_flash` segment, following is the order of section allocation: .init, .text, .rodata, .ctors, .dtors, extab and extabindex.

Variables are added in LCF and these can be used in application as well as internally in linker tool for computation.

Listing 5. Adding variables in LCF

```
_stack_addr = ADDR(stack)+SIZEOF(stack);
_stack_end  = ADDR(stack);
_heap_addr  = ADDR(heap);
_heap_end   = ADDR(heap)+SIZEOF(heap);
EXCEPTION_HANDLERS    = ADDR(exception_handlers_p0);
L2SRAM_LOCATION = 0x40000000;
```

Let us take a simple example to see how the allocation of variables to the respective sections take place.

Listing 6. C Source file

```
#include "MPC5675K.h"

int sdata_i = 10;
int sbss_i;
const char sdata2_array[] = "Hello";

__declspec(section ".rodata") const char rodata_array[40]="CodeWarior";
__declspec(section ".data") long bss_i;
__declspec(section ".data") long data_i = 10;

int main(void) {
  return sdata_i + sbss_i + sdata2_array[3] + data_i + bss_i + rodata_array[5];
}
```

NOTE Above is a hypothetical example built to provide clarity on variables and their allocation to sections. __declspec is used to forcefully place the variables into sections.

The objects are allocated to the sections as in the following table.

Table 1. Allocating Objects

Variable	Section	Address
sdata_i	.sdata	0x400004d8
sbss_i	.sbss	0x400004e8
sdata2_array	.sdata2	0x40000500
rodata_array	.rodata	0x00003938
bss_i	.bss	0x40000508
data_i	.data	0x400004d0

4 Relocating Code in ROM

To place data and code in a specific memory location there are two general steps that must be performed.

- Use pragma compiler directives to tell the compiler which part of the code is going to be relocated.
- Tell the linker where the code will be placed within the memory map using LCF definitions.

4.1 Relocating Function in ROM

To put code in a specific memory section it is needed first to create the section using the section pragma directive. In the following listing a new section called `.romsymbols` is created.

All the content in this section is going to be referenced in the LCF with the name `.romsymbols`. After defining a new section you can place code in this section by using the `__declspec()` directive.

In the following listing, `__declspec()` directive is used to tell the compiler that function `funcInROM()` is going to be placed in section `romsymbols`.

Create a stationary project for any target and add the following code to your `main.c` file before the main() function and have a call to this function.

Listing 7. Code to add in the main.c

```
#pragma section RX ".romsymbols" data_mode=far_abs
__declspec(section ".romsymbols") void funcInROM(int flag);    //Function Prototype
 void funcInROM(int flag){
if (flag > 0)
{
flag ++;
}
}
```

4.2 Placing Code in ROM

You have just edited a source file to tell the compiler which code will be relocated. Next, the LCF needs to be edited to tell the linker the memory addresses where these sections are going to be allocated.

First you need to define a new Microcontroller memory segment where new sections will be allocated.

You can have just one memory segment for all the new sections or one segment for each section.

4.2.1 Create New ROM Segment

Below you can find the memory segment of a LCF. Notice that the segment `internal_flash` has been edited and its length has been reduced by `0x10000` from its original size. This memory space is taken to create the new segment. In the following listing the new segment is called `myrom`, it will be located next to segment `internal_flash` and its length is going to be `0x10000`. You can calculate the address where segment code ends by adding its length plus the origin address.

Edit your LCF as shown in the following listing. Ensure you edit ROM target lcf.

Listing 8. Memory Segment of LCF

```
MEMORY
{
  resetvector:              org = 0x00000000,  len = 0x00000008
  init:                    org = 0x00000010,  len = 0x0000FFF0
  exception_handlers_p0:    org = 0x00010000,  len = 0x00010000
  internal_flash:           org = 0x00030000,  len = 0x001C0000
  myrom:                    org = 0x00220000,  len = 0x00010000

  internal_ram:             org = 0x40000000,  len = 0x0007C000
  heap  :                   org = 0x4007C000,  len = 0x00002000 /* z7_0 Heap start location */
  stack :                   org = 0x4007E000,  len = 0x00002000 /* z7_0 Start location for
Stack */
}
```

4.2.2 Create New ROM Section

The next step is to add the content of the new section into the Microcontroller memory segment you have reserved. This is done in the sections segment of the LCF.

The code below creates a new section called `.rom_symbols`, then the label `__ROM_SYMBOLS` points to the address where the section begins. Then `*(.romsymbols)` instruction is used to tell the linker that all the code referenced with this word is going to be placed in section `.rom_symbols`.

Finally you close the section telling the linker that this content is going to be located in segment myrom.

Edit your LCF as shown below.

Listing 9. Code to add your LCF.

```
.rom_symbols :
{
__ROM_SYMBOLS = . ;              #start address of the new symbol area
. = ALIGN (0x4);
```

```
*(.romsymbols)                                  #actual data matching pragma directives.
. = ALIGN (0x4);
} > myrom
```

Please note that in the MAP file you can notice newly created ROM section.

5 Relocating Code and Data in Internal RAM

Since it is not possible to write a variable in ROM, data must be relocated in RAM. Code can be also relocated in RAM. Another reason to relocate code in RAM is that it is twice as fast as in Flash.

5.1 Relocating Code and Data in Internal RAM

Create a new section using section pragma directive and __declspec directives as shown in the listing below.

Listing 10. Using pragma Directives to Define a Section

```
#pragma section  ".myCodeInRAM" data_mode=far_abs
__declspec(section ".myCodeInRAM")
struct {
unsigned char data0;
unsigned char data1;
unsigned char data2;
unsigned char data3;
unsigned char data4;
unsigned char data5;
unsigned char data6;
unsigned char data7;
} CTMData = { 0x82, 0x65, 0x77, 0x32, 0x84, 0x69, 0x83, 0x84 };

__declspec(section ".myCodeInRAM")  void funcInROM(int flag);
 void funcInROM(int flag){
if (flag > 0)
{
flag++;
}
}
```

5.2 Placing Code and Data in RAM

Placing code and data into RAM is more complicated. As the content in RAM cannot be saved when turning power off, you first need to save the code and data in flash and then make a copy to RAM in runtime.

Following are the steps to relocate code and data in a new RAM segment.

1. Create New RAM Segment
2. Placing Code and Data in RAM

5.2.1 Create New RAM Segment

As it was made for the new ROM segment, a piece of the user ram memory segment is taken to create a new memory segment called myram.

Edit your LCF as shown in Listing 5.

Listing 11. Memory Segment of LCF

```
MEMORY
{
    exception_handlers_p0:       org = 0x40000000,   len = 0x00001000
    pseudo_rom:                  org = 0x40001000,   len = 0x00006800
    init:                        org = 0x40007800,   len = 0x00000800
    internal_ram:                org = 0x40008000,   len = 0x00070000
    myram:                       org = 0x40078000,   len = 0x00004000
    heap  :                     org = 0x4007c000,   len = 0x00002000 /* Heap start location */
    stack :                      org = 0x4007e000,   len = 0x00002000 /* Start location for
Stack */
}
```

5.2.2 Create New RAM Section

The memory segment specifies the intended location in RAM. The code below shows a new section called `.my_ram` which is going to be linked in segment `.myram` but is going to be resident in the Flash memory address calculated by label `___CodeStart`. This label is intended to find the first address available in flash.

In the listing section `.app_text` the linker places in the segment code all the code and then the read only data. After this it sets a label called `__ROM_AT`. Section .data is allocated in the address pointed by this label.

Add the following code to LCF. You can put this code just after Placing data in RAM. The uninitialized and initialized data are placed in RAM.

Listing 12. Add this Code to LCF after Listing D.

```
___CodeStart = ___RAM_end;
.my_ram :
{
. = ALIGN (0x4);
___myRAMStart = .;
*(.myCodeInRAM)
___myRAMEnd = .;
. = ALIGN (0x4);
} > myram
```

6 Relocating Code and Data in External MRAM

Many times the internal RAM in the Microcontroller you are using is not enough for the application. For this reason it is needed to use external memories as part of the solution. The process to relocate code and

data in external memories is exactly the same as you did for internal RAM. The only difference is that the external device needs to be communicated by an interface controller.

7 Unique LCF Examples

This topic describes the following LCF examples.

- Configuring Linker File to Several ROM Blocks
- Place the Library File in the LCF
- Place Symbols in Specific Memory Location

7.1 Configuring Linker File to Several ROM Blocks

The following listing is an example to congiure linker file to several ROM blocks.

Listing 13. ROM IMAGE address = 0x3000

```
MEMORY{
 internal_flash:        org = 0x00003000,   len = 0x0010000
MyFlash:           org = 0x00041000,   len = 0x00000008
       //org should match the LOAD address
}
SECTIONS{
.text {} > internal_flash
.my_flash  ALIGN(0x08) LOAD(0x00041000) : {} > MyFlash
}
```

7.2 Place the Library File in the LCF

The following listing is an example to place the library file in the LCF.

Listing 14. Placing library file in LCF

```
GROUP : {
      .libcode (VLECODE) LOAD (0x00004000) : {
         Runtime.PPCEABI.VS.UC.a (.text)
       }
      .libconst:
       {
         Runtime.PPCEABI.VS.UC.a (.rodata)
       }
    } > lib_flash
```

NOTE For small data sections, ctors, dtors section it's not allowed to have different output section name.

7.3 Place Symbols in Specific Memory Location

For placing the symbols in specific memory location, user has to define the memory region (say Memory_to_store) in the lcf file and also define a new section (say .user_defined_section) then use the same section in the source file to place the symbol.

Listing 15. Example for initialized variable

```
In the source file:
#pragma section  <section_qualifier(R,RW)> ".user_defined_section"
__declspec(section ".user_defined_section") int temp = 5;
In the LCF file:
    GROUP : {
.user_defined_section :{}
    } > Memory_to_store             // Memory_to_store is the memory area where user want to
store
```

Listing 16. Example for uninitialized variable

```
In the source file:
#pragma section  ".user_defined_section"".data"
 __declspec(section ".user_defined_section") /* We cannot have an uninitialized section name
in The //uninitialized section must be paired with initialized section. */
__declspec(section ".user_defined_section") int temp;
In the LCF file:
    GROUP : {
.user_defined_section :{}
    } > Memory_to_store
```

7.4 How to Relocate Code in RAM

The following listing is an example to relocate the code in RAM.

Listing 17. Example to relocate the code in RAM

```
In the source file:
#pragma section ".myCodeInRAM" code_mode=far_abs

__declspec(section ".myCodeInRAM") int _add(int a , int b);

int main(void) {
  volatile int i = 0;
  volatile int total = 0;

    /* Loop forever */
  for (;;)  {
      total = _add(i , i);
    i++;
  }
}

__declspec(section ".myCodeInRAM") int _add(int a , int b)
{
    return a + b;
}
```

```
In the lcf file:
MEMORY
{
    ……….
     /* SRAM: 0x40000000 - 0x4000FFFF */
     internal_ram:          org = 0x40000000,   len = 0x0000D000
     myram:                 org = 0x4000D000,   len = 0x00001000
   ……..
}
……..
     GROUP : {
         .my_ram (VLECODE) : {          //VLECODE- if the code is the generated for VLE mode
            *(.myCodeInRAM)
           }
     } > myram
……..
```

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

Document Number: AN4497

18 June 2013