Freescale Semiconductor
Application Note

DRAFT

Document Number: ANxxxx

How to fill the unused memory gaps with a fill byte pattern in CodeWarrior for MCUs v10.5 for Qorivva/PX microcontroller family.

Contents

1. Introduction

CodeWarrior™ e200 build tools (b250+) released in CodeWarrior for MCU v 10.5 have a new fill pattern feature implemented. It allows filling of the unused memory and padding gaps with a defined byte pattern.

The application note describes how to use this feature. Additionally it includes some other tips related to filling the gaps of unused memory.

The PowerPC™ Embedded Application Binary Interface (EABI) requires data and code alignments. The alignment improves the overall performance but it causes the presence of the "gaps" of unused memory. There is now a way to control the content of padding patterns other then default 0x00. It is also possible to fill unused memory sections with a defined value.

The padding bytes within C structures/unions can be now changed from the default 0x00 to 0xFF using the new pragma: *fill_alignment_data_with_FF*.

freescale semiconductor

2. Structure/union alignment padding

Each structure member has its own native alignment based on the data types used. This is described in "PowerPC™ e500 Application Binary Interface User's Guide".
A structure as a whole is aligned based on the alignment requirements of the structure's largest member. Let's use the structure defined in Listing 1 as the example to illustrate the alignment gaps.

Listing 1. Example structure

```
typedef struct
{
  char      A;     // 1B byte
  long long B;     // 8B double word
  char      C;     // 1B byte
  int       D;     // 4B word
  char      E;     // 1B byte
  short     F;     // 2B half word
}tTestStruct;
```

The memory view Figure 1 shows the presence of padding alignment bytes due to the native alignment. The size of this aligned structure is 32 bytes.

Figure 1. Default alignment example structure memory allocation

	0						7
A	00	00	00	00	00	00	00
B	B	B	B	B	B	B	B
C	00	00	00	D	D	D	D
E	00	F	F	00	00	00	00

The padding bytes are zeros by default. It can be changed to 0xFF by adding the pragma below into the source files or the prefix header file:

```
#pragma fill_alignment_data_with_FF on
```

or pass it via the command line option:

```
-pragma "fill_alignment_data_with_FF on"
```

The Figure 2 shows the memory view of the example structure that includes the 0xFF padding pattern.

Figure 2. Structure memory allocation with 0xFF padding pattern

	0						7
A	FF	FF	FF	FF	FF	FF	FF
B	B	B	B	B	B	B	B
C	FF	FF	FF	D	D	D	D

E	FF	F	F	FF	FF	FF	FF

In order to completely avoid padding between struct members **#pragma pack (1)** can be used. This pragma can align data to use less storage even if the alignment might affect program performance or does not conform to the target platform's application binary interface. The size of this structure is 17 in contrast to the default aligned structure (see the Figure 3).

Figure 3. Structure memory allocation with #pragma pack (1) enabled

0 7

A	B	B	B	B	B	B	B
B	C	D	D	D	D	E	F
F							

3. Alignment padding within a linker section or across multiple sections

Similar to C structs other data objects have their own native alignment requirements. C-function code is aligned with alignment that can be either configured globally (directly in IDE – see Figure 4 **-func_align** compiler command line option) or locally for specific functions only using **#pragma function_align**.

Object alignment also causes gaps of unused memory between objects.
The new linker feature can fill such gaps with a defined short word pattern. The short word pattern is a two-byte decimal or hexadecimal pattern. E.g. to fill the memory area with 0xFF the pattern is 0xFFFF.

The linker syntax for filling alignment pattern within a single linker section is described in the Listing 2 below.

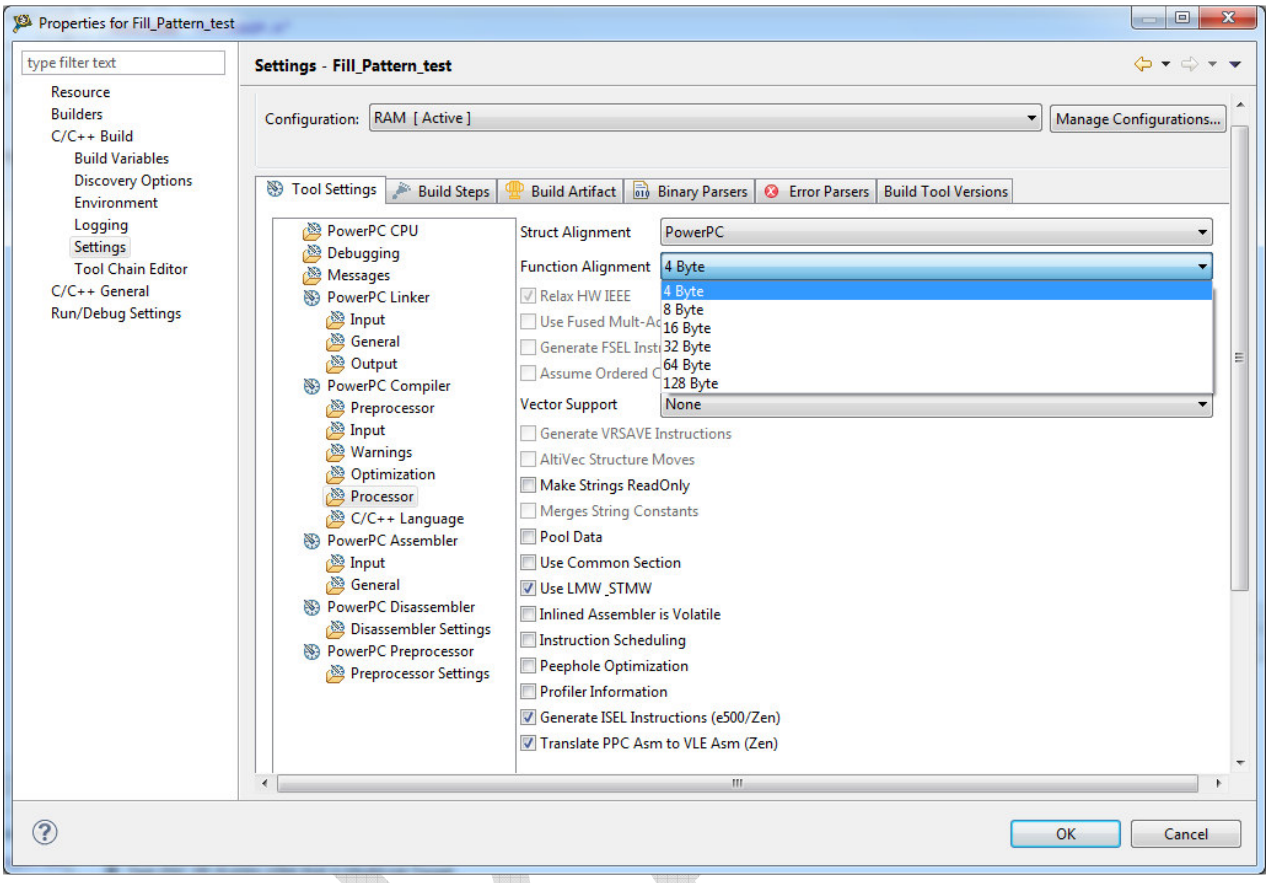
Listing 2. Single linker section alignment pattern fill (syntax + example)

```
section-spec =
output-section-name ":"["(" input-type ")"] [address-modifiers]
"{"
[( input-section-spec )*]
"}" [= fill-shortnumber] [(">") memory-area-symbol ]


The example:

.rodata (CONST) : {
  *(.rdata)
  *(.rodata)
 }  = 0xBEEF > internal_flash
```

Figure 4. The function alignment global IDE settings in CW for MCUs v10.x (Eclipse)



The linker syntax for filling alignment pattern within the objects of the same section as well as across multiple linker sections is described below in Listing 3.

Listing 3. Multiple linker section alignment pattern fill (syntax + example)

```
"GROUP" address-modifiers ":"
"{"
  (section-spec )*
"}" [= fill-shortnumber ] [ ">" memory-area-symbol ]

The example:

GROUP  : {
  .intc_sw_isr_vector_table_p0 ALIGN (2048) : {}
  .text : {}
  .text_vle (VLECODE) : {
     *(.text)
     *(.text_vle)
  }
  .rodata (CONST) : {
     *(.rdata)
     *(.rodata)
```

```
  }
  .ctors : {}
  .dtors : {}
   extab : {}
   extabindex : {}
}  = 0xBEEF > internal_flash
```

4. Filling the unused memory space of a linker section

The linker feature above can also be used to fill the unused memory with a predefined short word pattern. It requires some additional linker command file (.lcf) modifications. The linker command file should create a gap the size of the remaining unused memory space. See the fragment of .lcf file example in Listing 4.

Listing 4. Example .lcf file that fills the unused memory of with the pattern 0xABCD

```
MEMORY
{
…
   my_memory:       org = 0x00001000,   len = 0x00001000   // custom memory space
   internal_flash:  org = 0x00002000,   len = 0x0007E000
 ...
}
…
SECTIONS
{
…
   // LOAD directive to avoid ROM Image to be generated
  .my_memory LOAD (ADDR(my_memory)) :
  {
…
     // create a memory gap of remaining unused memory area
      . =  SIZEOF(my_memory) + ADDR(my_memory);
   } = 0xABCD >  my_memory
```

The unused space is calculated from the memory area start address “ADDR(my_memory)” and its size “SIZEOF(my_memory)”. The end memory address is calculated and assigned to the current position pointer (“.” character). This causes the gap to be created.

> **NOTE:** This approach does not work for a main flash section – the section where the ROM image is placed. The main flash section (usually Internal_Flash) requires different calculation of unused memory (see Listing 6)

The unused memory byte pattern should appear in the generated binary .bin and .mot (s-record) files. See the fragment of generated s-record which matches with the example above in Listing 5.

Listing 5. Example .lcf file that fills the unused memory of with the pattern 0xABCD

```
...
  S319000001000 ABCD ABCDABCDABCDABCDABCDABCDABCDABCDABCD 73
```

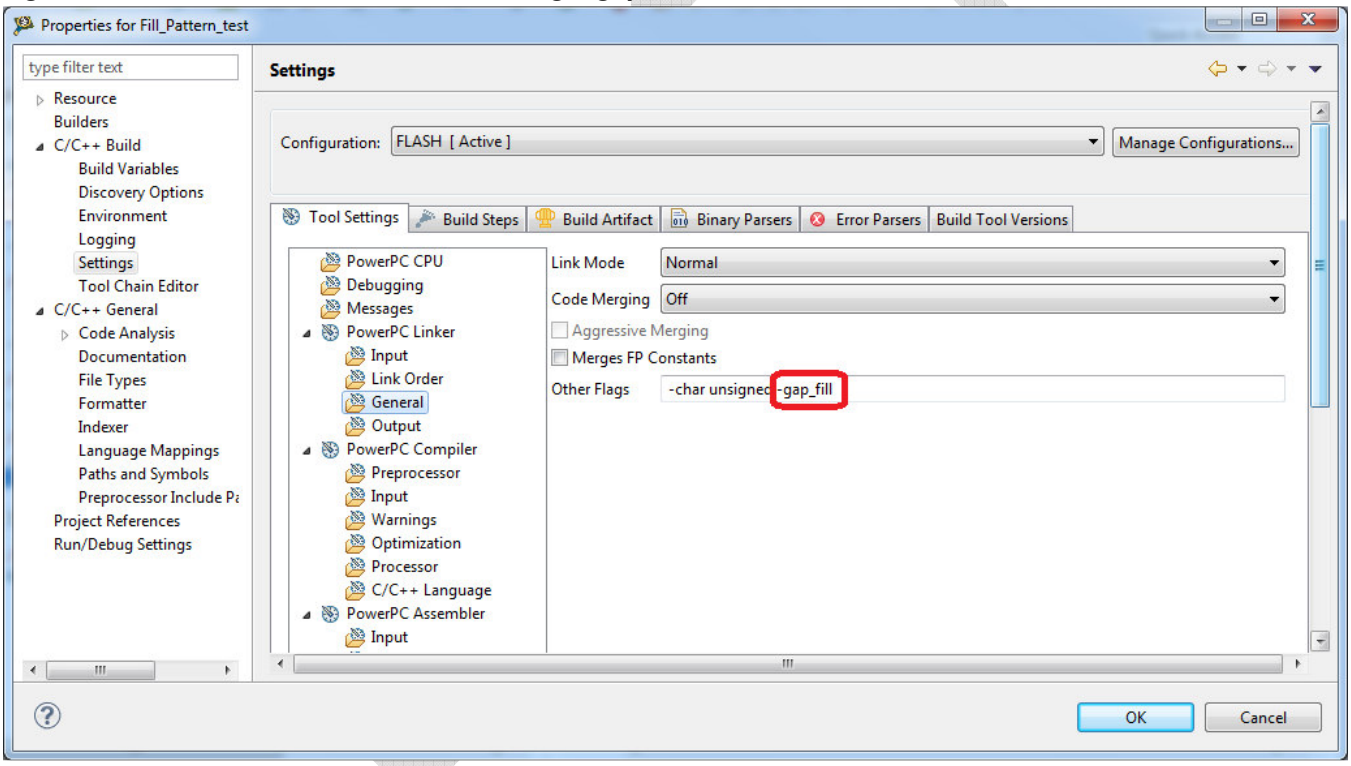
```
S31900001014 ABCD ABCDABCDABCDABCDABCDABCDABCDABCDABCD 5F
...
```

5. Filling of unused Flash memory space with a pattern

If the entire unused Flash space needs to be filled by a predefined pattern then:

- Make sure there is no gap in MEMORY Flash areas in the .lcf file. Such a gap cannot be filled.
- Define a separate unused memory section within each memory SECTION definition in the .lcf file except main memory block (usually named “Internal_Flash”).
- There should be one extra SECTION that includes all the remaining unused flash space created for the main memory block.
- Add the linker switch “-gap_fill” which is used to fill the alignment gaps and the gaps created by LOAD address modifiers (see Figure 5).

Figure 5. CodeWarrior for MCU v10.x adding “-gap_fill” linker switch



The extra section requires the address correction calculation in order to reflect ROM image size and the alignments.

The calculation of ROM image size requires two addresses:

- ROM image start address (ROM_IMG_START symbol in example below)
- ROM image end address (ROM_IMG_END)

ROM image data size is calculated as:

```
ROM_DATASIZE = ROM_IMG_END - ROM_IMG_START;
```

NOTE: The fill pattern feature does not impact the .elf file. If a default debug session starts the fill pattern areas are not programmed. In order to program the image including the fill pattern blocks into Flash the .mot (S-record) file or .bin needs to be programmed into flash e.g. using the Flash Programmer/Target Task tool.

Listing 6. Example of .lcf file that fills the entire unused Flash memory with the predefined patterns

```
MEMORY
{
    /*FLASH: 0x00000000 - 0x00003FFFF*/

    /* Fixed location required for RCHW and program entry point. */
    resetvector:           org = 0x00000000,   len = 0x00000008

    /* Contains initializations from __ppc_eabi_init.c,
     * MPC56xx_HWInit.c, MPC56xx_init_*.c and the entry point (__startup).
     */
    init:                  org = 0x00000008,   len = 0x00000FF8

    /* Contains interrupt branch tables for both core and INTC module
     * and the ISR handlers code. Note, since the vector base address field
     * of IVPR is defined within the range [0:19] the table must be loaded
     * at an address aligned to 4K boundary.
     */
    exception_handlers:    org = 0x00001000,   len = 0x00001000

    /* Space allocated for user code and device initialization.
     * ROM Image address should be set with the start address of this
     * segment in order to instruct the runtime to initialize the
     * static variables. All the section above are ignored for this action.
     * Please see -romaddr linker option.
     */
    internal_flash:        org = 0x00002000,   len = 0x0003E000

    /* Space allocated for both code and variables in order to use the memory
     * more efficiently.*/
     * SRAM: 0x40000000 - 0x40004FFF
     */
    internal_ram:       org = 0x40000000,   len = 0x00003000 /* 12K */
    heap:               org = 0x40003000,   len = 0x00001000 /* 4K Heap */
    stack:              org = 0x40004000,   len = 0x00001000 /* 4K Stack */
}

/* This will ensure the rchw and reset vector are not stripped by the linker */
FORCEACTIVE { "bam_rchw" "bam_resetvector" }

SECTIONS
```

```
{
    .__bam_bootarea LOAD (ADDR(resetvector)): {} > resetvector

    GROUP : {
      /* Section used for initialization code: __ppc_eabi_init.c,
       * MPC56xx_HWInit.c, MPC56xx_init_*.c and the entry point (__startup).
       */
      .init LOAD(ADDR(init)) : {}
      .init_vle (VLECODE) LOAD(_e_init) : {
        *(.init)
        *(.init_vle)
      }
      /* create a gap of unused memory area "init" which is filled by the pattern */
      .unused_init LOAD(_e_init_vle): {
        . = ADDR(init) + SIZEOF(init);
      }
    } = 0xAAAA > init

    GROUP : {
      /* Fixed length of 0x800 is required for core interrupt branch table.
       * Core IVOR branch table entries must be aligned to 16 bytes.
       */
      .ivor_branch_table (VLECODE) LOAD(ADDR(exception_handlers)) ALIGN (0x10) : {}

      /* Note if intc_hw_branch_table is used it MUST be loaded at the IVPR + 2KB (0x800).
       * For this device the intc_hw_branch_table should contain entries aligned to 4
       * bytes.
       */
      .intc_hw_branch_table (VLECODE) LOAD(_e_ivor_branch_table) ALIGN(0x800) : {}

      /* ISR handlers code.*/
      .__exception_handlers  (VLECODE) LOAD(_e_intc_hw_branch_table) : {}

      /* create a gap of unused memory area "exception_handlers" which is filled by the
       * pattern
       */
      .unused_exception LOAD(_e___exception_handlers) : {
        . = ADDR(exception_handlers) + SIZEOF(exception_handlers);
      }
    }  = 0xBBBB> exception_handlers

    /* User application code and data. */
    GROUP : {
      .text : {}
      .text_vle (VLECODE) ALIGN(0x08): {
         *(.text)
         *(.text_vle)
       }
       .rodata (CONST) : {
         *(.rdata)
         *(.rodata)
       }
```

```
        .ctors : {}
        .dtors : {}
        extab : {}
        extabindex : {}
        /* save the start address of ROM data image */
        ROM_IMG_START=.;
    } = 0xCCCC > internal_flash

    GROUP : {
        /* This section is used in INTC SW mode to store the interrupt handlers array.
         * It should be aligned to 2K since VTBA = INTC_IACKR[0:20].
         */
        .__uninitialized_intc_handlertable ALIGN(0x800) : {}
        .data   : {}
        .sdata  : {}
        .sbss   : {}
        .sdata2 : {}
        .sbss2  : {}
        .bss    : {}
        /* save the last ROM data image address */
        ROM_IMG_END = ROMADDR(.bss);
     } = 0xCCCC > internal_ram   //This fills the ROM image gaps


    /* calculation of ROM data image size */
    ROM_DATASIZE = ROM_IMG_END - ROM_IMG_START ;

    .unused_int_flash :
      {
        /* gap size needs to be adjusted to reflect ROM data image size */
        . = ADDR(internal_flash) + SIZEOF(internal_flash) - ROM_DATASIZE;
     }= 0xEEEE > internal_flash
}
```

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: ANxxxx

20 July 2016