Freescale Semiconductor, Inc.
Application Note

Document Number: AN5163
Rev. 0, 07/2015

Load Position-Independent Code (PIC) on a Kinetis Platform Using the IAR EWARM Compiler

1. Introduction

This document provides guidance for how to implement and load position-independent code (PIC) for the Kinetis Microcontroller platform using IAR EWARM compiler.

Position-independent code (PIC) is a body of program code that executes properly regardless of its absolute address because it is placed somewhere in the memory. For a flash-based microcontroller like the Kinetis ARM® Cortex®-M product family, position-independent technology can be used for the bootloader to download and load an application code to any address to run. This provides the flexibility to have two or more application images reside in the flash memory and the user can choose one of them to be active without modifying their link address.

The key initiative to keep two or more application images in flash memory is to make sure at least one image is ready to use, once a power loss or communication interrupt event occurs during a firmware upgrade in field. For microcontrollers without a swappable dual bank flash, this is the primary choice to implement a reliable over-the-air (OTA) upgrade.

Figure 1 shows three different bootloader solutions.

Contents

1. Introduction 1
2. Implement a position-independent application 3
2.1. Limitations 6
3. Implement a PIC application Loader 7
4. Demo guide 7
4.1. Demo memory map 8
4.2. Build and download Loader project 8
4.3. Build and download Application project 9
4.4. Check LED and terminal output 9
5. Revision History 10

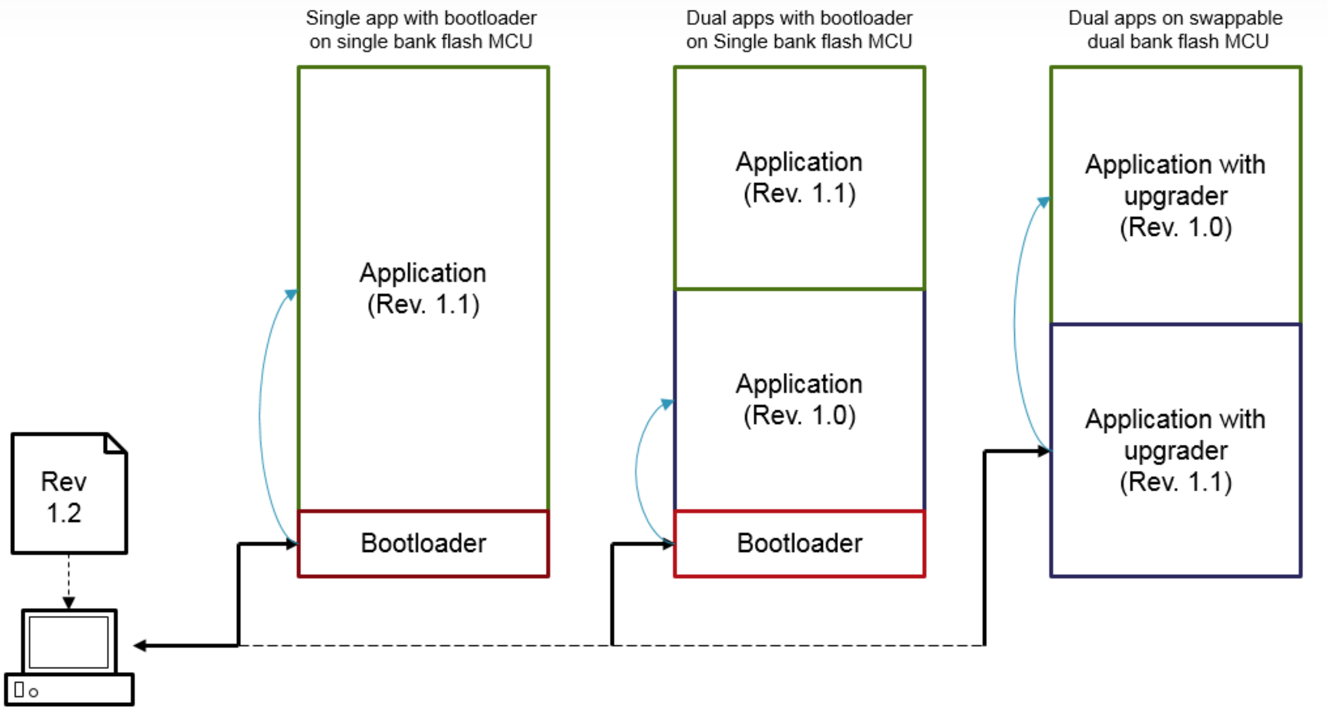
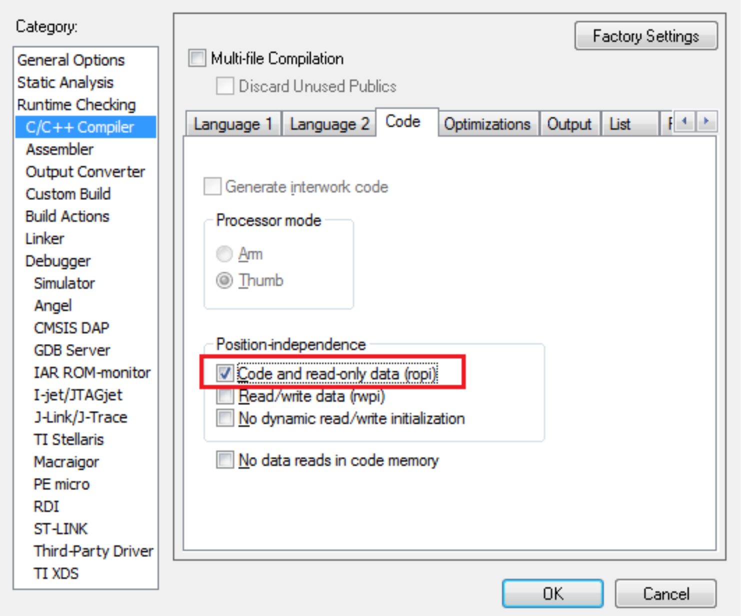

Single app with bootloader on single bank flash MCU
Dual apps with bootloader on Single bank flash MCU
Dual apps on swappable dual bank flash MCU
Application (Rev. 1.1)
Bootloader
Application (Rev. 1.1)
Application (Rev. 1.0)
Bootloader
Application with upgrader (Rev. 1.0)
Application with upgrader (Rev. 1.1)
Rev 1.2

Figure 1. Different solutions for bootloader

2. Implement a position-independent application

A position-independent application project is a little different from a traditional application. The binary image built should not contain an absolute address jump. Instructions that see specific memory addresses must be replaced with equivalent program counter relative instructions. This important step is usually done by a complier option. For the IAR EWARM compiler, the --*ropi* option makes the compiler generate code that uses PC-relative references to address code and read-only data. This includes constant data and data initializers. Typically, everything in flash is relative referenced. Another option, –*rwpi*, is used for read-write data position-independent control, which is not used or discussed in this article. The settings for --*ropi* in IAR EWARM are shown in the *Figure 2*. Ensure the user libraries that are linked together are all compiled with the --*ropi* option.



□ Multi-file Compilation
□ Discard Unused Publics

□ Generate interwork code

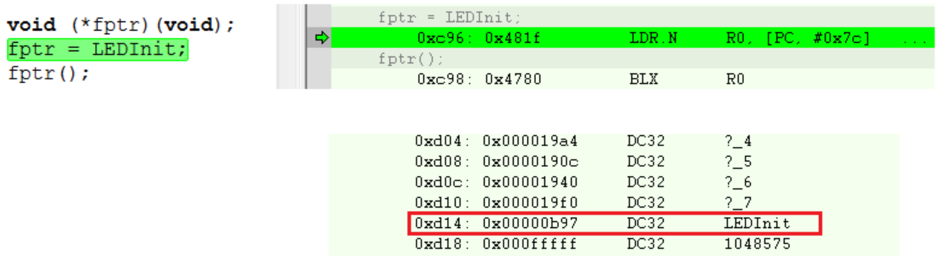
Processor mode
○ Arm
● Thumb

Position-independence
☑ Code and read-only data (ropi)
□ Read/write data (rwpi)
□ No dynamic read/write initialization

□ No data reads in code memory

Figure 2. Enable --*ropi* compiler option

Below are two examples that demonstrate a piece of function pointer usage code compiled with and without the --*ropi* option. The assembler code, shown in Figure 3, retrieves the absolute function address from a literal pool. The processor jumps to 0xB97 for whatever program been loaded. This does not work if the 0xB97 address does not have the LEDInit() assembler code any longer. The assembler code shown in Figure 4 calculates the function address according to current PC value. Once an image is loaded to another address, the run-time value of the function pointer is updated respectively.

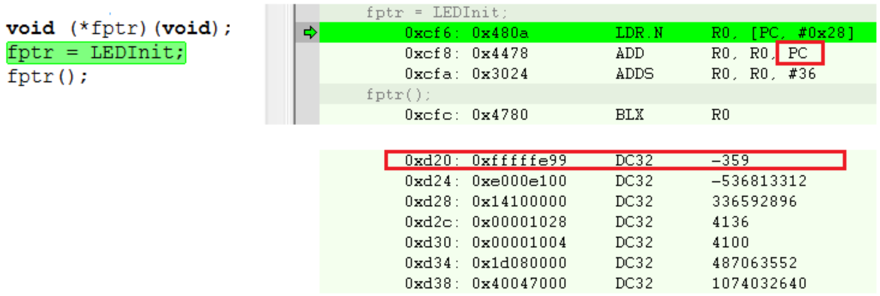


```
void (*fptr)(void);
fptr = LEDInit;
fptr();
```

```
fptr = LEDInit;
    0xc96: 0x481f        LDR.N      R0, [PC, #0x7c]   ...
fptr();
    0xc98: 0x4780        BLX        R0
```

```
0xd04: 0x000019a4    DC32    ?_4
0xd08: 0x0000190c    DC32    ?_5
0xd0c: 0x00001940    DC32    ?_6
0xd10: 0x000019f0    DC32    ?_7
0xd14: 0x00000b97    DC32    LEDInit
0xd18: 0x000fffff    DC32    1048575
```

Figure 3. C and Disassembler compiled without --*ropi* option



```
void (*fptr)(void);
fptr = LEDInit;
fptr();
```

```
fptr = LEDInit;
    0xcf6: 0x480a        LDR.N      R0, [PC, #0x28]
    0xcf8: 0x4478        ADD        R0, R0, PC
    0xcfa: 0x3024        ADDS       R0, R0, #36
fptr();
    0xcfc: 0x4780        BLX        R0
```

```
0xd20: 0xfffffe99    DC32    -359
0xd24: 0xe000e100    DC32    -536813312
0xd28: 0x14100000    DC32    336592896
0xd2c: 0x00001028    DC32    4136
0xd30: 0x00001004    DC32    4100
0xd34: 0x1d080000    DC32    487063552
0xd38: 0x40047000    DC32    1074032640
```

Figure 4. C and Disassembler compiled with --*ropi* option

Besides using the --*ropi* option for C/C++ code, any bootstrap code or mixed assembler modules used with ARMv6M or ARMv7M assembler language should not use absolute address branch instructions. The typical pseudo instructions are shown in Example 1.

Example 1. Pseudo assembler to jump to absolute address label

```
LDR     PC,     =Label

;or

LDR     R0,     =Label
BLX     R0
```

The user should SystemInit() and __iar_data_init3() in the beginning of main() to make sure basic hardware and C environment are established. This is shown below in Example 2.

Example 2. startup_MKL25Z4_app.s

```
                    MODULE  ?cstartup
                    SECTION CSTACK:DATA:NOROOT(3)
                    SECTION .intvec:CODE:NOROOT(2)

                    EXTERN  main
                    PUBLIC  vector_table

                    DATA

__vector_table
                    DCD     sfe(CSTACK)
                    DCD     main              ;load main()

                    DCD     NMI_Handler       ;NMI Handler
                    ...
```

For ARM Cortex-M based architecture, the NVIC vector table contains 48 or 256 entries for exception and interrupt. The value placed into the vector table is typically a function pointer, which is an absolute address determined at the link stage, except for the first one used for CSTACK. This includes the second vector for the first address load to the PC for out-of-chip RESET. To make this position-independent, each entry should be recalculated according to the image’s run-time load address. This step is done by the loader. The loader also assigns an offset to the NVIC vector table base address (SCB_VTOR) before starting the application. If the application wants to relocate the vector table to RAM, it needs to determine where it should be placed by reading the SCB_VTOR register, then copying the right content from flash to RAM.

NOTE

Though the application is built with a position-independent option, all symbols have absolute address assigned at the link stage. Generally, the application should be linked to 0x0 flash address for easy vector table content offset calculation. For example, the function do_things () may have its address assigned by compiler: 0x000003B1, but the actual run-time address of do_things () is position-independent, and calculated according the current PC value dynamically. Therefore, it may be 0x000043B1 or 0x000083B1 depending on the load address. A code example used to check the run-time address of a function is shown below in Example 3.

Example 3. Code to check run-time address of a function

```
printf(“the run-time address of do_things() is 0x%x”, do_things);
```

2.1. Limitations

When the --*ropi* option is used, these limitations apply to C/C++ code:

- C++ constructions cannot be used.
- The object attribute *__ramfunc* cannot be used.
- Pointer constants cannot be initialized with the address of another constant, a string literal, or a function. However, writable variables can be initialized to constant addresses at runtime. See Example 4 and Example 5 for examples.

Example 4. Pointer constant initialized with the address of another constant (NOT allowed)

```
const uint32_t CONSTDATA = 100;
const uint32_t * const pointer_in_flash = &CONSTDATA;

const hwtimer_devif_t kPitDevif =
{
    HWTIMER_SYS_PitInit,
    HWTIMER_SYS_PitDeinit,
    HWTIMER_SYS_PitSetDiv,
    HWTIMER_SYS_PitStart,
    HWTIMER_SYS_PitStop,
    HWTIMER_SYS_PitGetTime,
};
```

Example 5. Pointer variable initialized with the address of constant (Allowed)

```
const uint32_t CONSTDATA = 100;
const uint32_t *pointer_in_RAM = &CONSTDATA;

hwtimer_devif_t kPitDevif =
{
    HWTIMER_SYS_PitInit,
    HWTIMER_SYS_PitDeinit,
    HWTIMER_SYS_PitSetDiv,
    HWTIMER_SYS_PitStart,
    HWTIMER_SYS_PitStop,
    HWTIMER_SYS_PitGetTime,
};
```

3. Implement a PIC application loader

A PIC application loader is a piece of small binary code that resides at the beginning of flash memory and runs out-of-chip RESET. It usually contains user logic to decide whether it should receive a new image from host, or load an appropriate application image instead.

The special design of a position-independent application loader means it must modify received vector content, plus a predefined offset, before programming to one or more predefined flash addresses. For dual or multiple images, the loader is also responsible for version and integrity tracking in order to load the newest acceptable image for running. The typical workings of a dual image loader is shown in a flowchart in *Figure 5*.

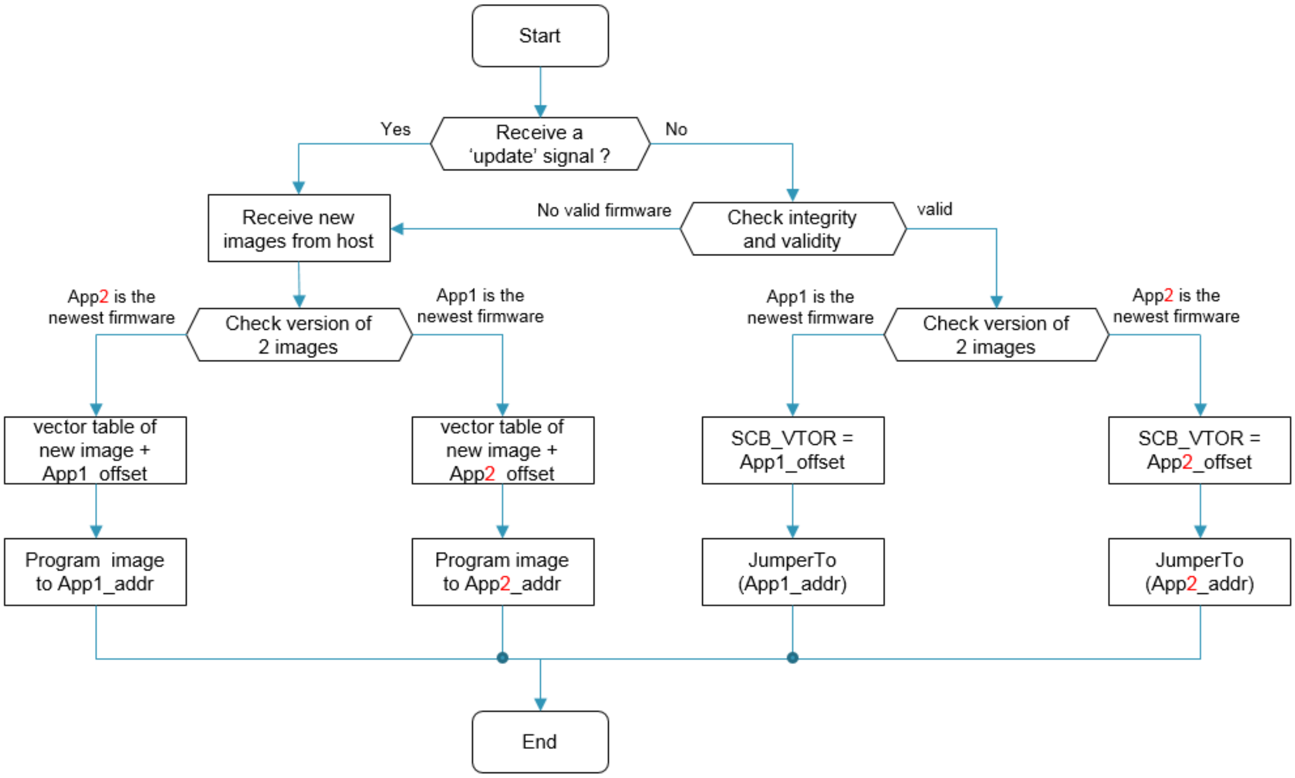

Start
Receive a 'update' signal ?
Yes
No
Receive new images from host
No valid firmware
Check integrity and validity
valid
App2 is the newest firmware
Check version of 2 images
App1 is the newest firmware
App1 is the newest firmware
Check version of 2 images
App2 is the newest firmware
vector table of new image + App1_offset
vector table of new image + App2_offset
SCB_VTOR = App1_offset
SCB_VTOR = App2_offset
Program image to App1_addr
Program image to App2_addr
JumperTo (App1_addr)
JumperTo (App2_addr)
End

Figure 5. Typical working flowchart for dual image loader

4. Demo guide

A demo built on the FRDM-KL25Z Freescale Freedom development platform is provided together with this application note to demonstrate position-independent code in dual images loader solution. The demo contains two standalone projects, which are the application and loader. Both projects are developed on IAR EWARM v7.40 and a simplified template copy from the Kinetis software development kit (SDK) version 1.2.0. Though it is built on a specific hardware platform, it is easy to reuse on other Kinetis platforms with minimal effort.

The loader duplicates the raw binary to two application logic banks, and loads one of them according to the GPIO state on the freedom board. The raw binary, built with position-independent instructions, can run at a different memory location to light the on-board LED as red, green, or orange. Use printf as the run-time address information for the terminal console.

4.1. Demo memory map

The FRDM-KL25Z Freescale Freedom development platform with MKL25Z128VLK4 has 128 KB flash memory, and can be divided into four regions for the demo’s purpose. The raw binary region is used to demo a received complete image from the host, so the user can program the application through a debug probe. See Figure 6 for address range of each region.

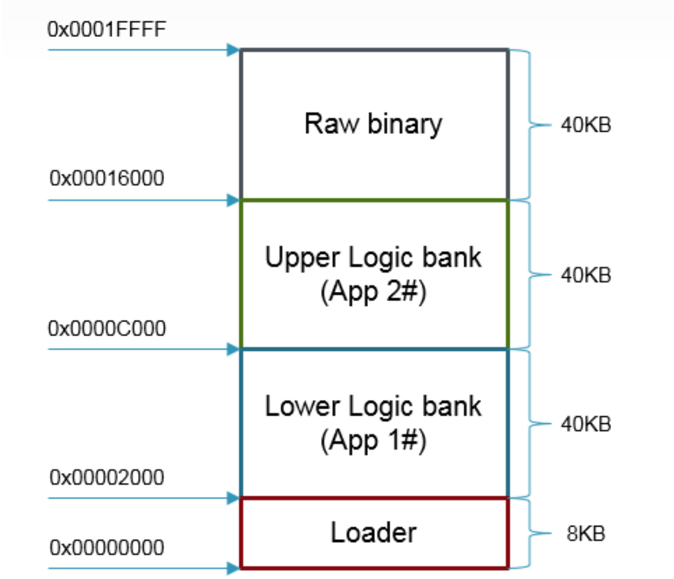


Figure 6. Demo memory map

4.2. Build and download Loader project

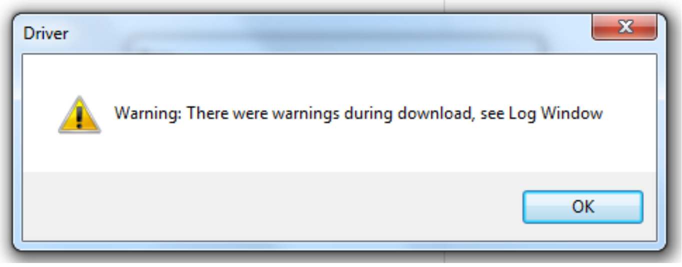
These are the instructions to build and downloader Loader project:

1. Unzip to folder and open ROPI.eww workspace file with IAR EWARM 7.40 or later.
2. Select the “Loader - Debug” project and perform a “Rebuild All” action.
3. Power the FRDM-KL25Z Freescale Freedom development platform through a J7 SDA USB Mini-connector and ensure a P&E Micro OpenSDA Debug Driver is enumerated in Device Manager.
4. Click ▶ “Download and debug” button.
5. After finished, click ✕ “Stop Debugging” button

4.3. Build and download Application project

These are the instructions to build and download Application project:

1. Make sure the <abs_offset> item in Application.board is filled with the 0x16000 address. This forces the IAR flashloader program image to a 0x16000 address instead of its linker base address.
2. Select “Application - Debug” project and perform a “Rebuild All” action.
3. Power the FRDM-KL25Z Freescale Freedom development platform through J7 SDA USB Mini-connector and ensure a P&E Micro OpenSDA Debug Driver is enumerated in Device Manager.
4. Click “Download and debug” button on debug control panel. Ignore the prompt warning window shown in Figure 7.



Warning: There were warnings during download, see Log Window

Figure 7. Prompt to alert debug information not matching

5. After finished, click ✖ “Stop Debugging” button.

4.4. Check LED and terminal output

1. Power FRDM-KL25Z Freescale Freedom development board through the J7 SDA USB Mini-connector and ensure an OpenSDA – CDC Serial Port Driver is enumerated in Device Manager.
2. Open COMx with a terminal console software, 115200, N, 8, 1.
3. Press SW1-RESET. The LED begins to blink and the console prompts “Application is running at lower flash logic bank”.
4. Short J9-13 (PTB4) and J9-14 (GND), then press SW1-RESET. The LED begins to blink and the console prompts “Application is running at upper flash logic bank”.

5. Revision History

This table summarizes revisions to this document.

Table 1. **Revision history**

Revision Number	Date	Substantive changes
0	07/2015	Initial Release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.