
Freescale Semiconductor

Interrupt handling with KSDK and Kinetis Design Studio

By: Jorge Gonzalez / Technical Information Center

Freescale Semiconductor

About this document

The Kinetis Software Development Kit (KSDK) is intended for rapid evaluation and development with Kinetis family MCUs. Besides of the peripheral drivers, the hardware abstraction layer and middleware stacks, the platform provides a robust interrupt handling mechanism.

This document explains the implementation and handling of interrupts when using KSDK in baremetal mode and when using MQX RTOS. Kinetis Design Studio IDE was used as reference, but the concepts should apply for any particular IDE supported by KSDK.

Software versions

The contents of this document are valid for the latest versions of Kinetis SDK and Kinetis Design Studio by the time of writing, listed below:

- KSDK v1.2.0
- KDS v3.0.0

Content

- 1. GLOSSARY**
- 2. CONCEPTS AND OVERVIEW**
 - 2.1 Interrupt Manager**
 - 2.2 Vector table location**
 - 2.3 Interrupt priorities**
- 3. KSDK INTERRUPT HANDLING**
 - 3.1 Baremetal interrupt handling**
 - 3.2 MQX RTOS interrupt handling**
 - 3.3 Operating System Abstraction layer (OSA)**
- 4. KDS AND PROCESSOR EXPERT CONSIDERATIONS**
 - 4.1 KSDK baremetal**
 - 4.2 KSDK baremetal + Processor Expert**
 - 4.3 MQX for KSDK**
 - 4.4 MQX for KSDK + Processor Expert**
- 5. REFERENCES**

Freescale Semiconductor

1. GLOSSARY

- KSDK** *Kinetis Software Development Kit*: Set of peripheral drivers, stacks and middleware layers for Kinetis microcontrollers.
- KDS** *Kinetis Design Studio*: Integrated Development Environment (IDE) software for Kinetis MCUs.
- API** *Application Programming Interface*: Refers to the set of functions, methods and macros provided by the different layers of the KSDK platform.
- ISR** *Interrupt Service Routine*: Also called “Interrupt Handler Routine” or simply “Interrupt Handler”, is a code routine in charge of taking action when a determined interrupt is triggered.
- NVIC** *Nested Vector Interrupt Controller*: Standard module of ARM® Cortex® cores which is in charge of handling interrupts and exceptions at the hardware level by enabling/disabling interrupts, assigning priorities, nesting interrupts among other functions.
- CMSIS** *Cortex Microcontroller Software Interface Standard*: Standard interface for ARM® Cortex® based MCUs. This hardware layer provides a set of functions and methods to interface directly to the MCU core.

VECTOR NUMBER Number assigned to an interrupt in relation to the core. This number considers the ARM® Cortex® system interrupts, which are 16 in total. In Kinetis Reference Manuals, the vector numbers are listed in the Vector Assignment Table under the **Vector** column.

IRQ NUMBER Assigned non-core interrupt number. This number only considers the peripheral modules interrupts, which are seen as ‘external’ by the NVIC module. In Kinetis Reference Manuals, the IRQ Numbers are listed in the Vector Assignment Table under the **IRQ** column and are equal to (VECTOR NUMBER – 16).

Address	Vector	IRQ	NVIC non-IPR	NVIC IPR	Source module	Source description
ARM Core System Handler Vectors						
0x0000_0000	0	–	–	–	ARM core	Initial Stack Pointer
0x0000_0004	1	–	–	–	ARM core	Initial Program Counter
						(PendableSrvReq)
0x0000_003C	15	–	–	–	ARM core	System tick timer (SysTick)
Non-Core Vectors						
0x0000_0040	16	0	0	0	DMA	DMA channel 0 transfer complete

Figure 1.1- Interrupt vector assignments table example

Freescale Semiconductor

2. CONCEPTS AND OVERVIEW

This section provides general information and concepts about KSDK interrupt handling system.

2.1 Interrupt Manager

The KSDK Interrupt Manager provides APIs to enable and disable individual interrupts or enable/disable them in a global way. In addition it allows registration of interrupt handlers defined by the user application. The interrupt manager is used by the peripheral drivers for enabling or disabling particular interrupts, but it can also be used directly by the application using the APIs provided.

The interrupt manager consists of two files: “fsl_interrupt_manager.c” and “fsl_interrupt_manager.h”.

fsl_interrupt_manager.c: This source file is built as part of KSDK platform library and it provides the definition of the next APIs:

- **INT_SYS_InstallHandler (irqNumber, handler):** Allows the application to replace or install an interrupt handler for the specified IRQ number. This function requires the vectors to be located in RAM since the ISR is installed directly to the vector table.
- **INT_SYS_EnableIRQGlobal (void):** Enables the global interrupt by calling the `__enable_irq()` API from the CMSIS file `core_cmFunc.h`.
- **INT_SYS_DisableIRQGlobal (void):** Disables the global interrupt by calling the `__disable_irq()` API from the CMSIS file `core_cmFunc.h`.

fsl_interrupt_manager.h: This file has to be included to call any of the functions defined in “fsl_interrupt_manager.c”. It also defines 2 relevant APIs:

- **INT_SYS_EnableIRQ (irqNumber):** Enables the interrupt for the specified IRQ number by calling the API `NVIC_EnableIRQ ()` from the CMSIS core header file⁽¹⁾.
- **INT_SYS_DisableIRQ (irqNumber):** Disables the interrupt for the specified IRQ number by calling the API `NVIC_DisableIRQ()` from the CMSIS core header file⁽¹⁾.

(1) NVIC related APIs are defined in the CMSIS Core Peripheral Access Layer Header File: `core_cm4.h` or `core_cm0plus` depending on the Kinetis family.

Freescal Semiconductor

2.2 Vector table location

The interrupt vector table is located by default in Flash memory at compile time, by pointing either to the weak handlers found in the CMSIS startup code file or the Peripheral Drivers ISR entry functions defined throughout the project source files.

The KSDK delivered linker files are prepared to relocate vector table to RAM, by defining the argument `__ram_vector_table__ = 1` and passing it to the linker, which is explained in Chapter 4 for KDS IDE. With this symbol defined, the vector table will be copied from Flash to RAM during startup inside of the function `init_data_bss()` from file `startup.c`:

```
extern uint32_t __VECTOR_TABLE[];
extern uint32_t __VECTOR_RAM[];
extern uint32_t __RAM_VECTOR_TABLE_SIZE_BYTES[];
uint32_t __RAM_VECTOR_TABLE_SIZE = (uint32_t)(__RAM_VECTOR_TABLE_SIZE_BYTES);
#endif

if (__VECTOR_RAM != __VECTOR_TABLE)
{
    /* Copy the vector table from ROM to RAM */
    for (n = 0; n < ((uint32_t)__RAM_VECTOR_TABLE_SIZE)/sizeof(uint32_t); n++)
    {
        __VECTOR_RAM[n] = __VECTOR_TABLE[n];
    }
    /* Point the VTOR to the position of vector table */
    SCB->VTOR = (uint32_t)__VECTOR_RAM;
}
else
{
    /* Point the VTOR to the position of vector table */
    SCB->VTOR = (uint32_t)__VECTOR_TABLE;
}
```

Symbols defined in linker file

Figure 2.1 - Copy of vector table to RAM at startup

NOTE

The `__ram_vector_table__` symbol is not required for Processor Expert projects. The location of vectors in Flash/RAM is handled by the generated linker file, which is controlled by the CPU component build options as explained in Chapter 4.

Freescal Semiconductor

2.3 Interrupt priorities

The interrupt priorities are determined by the application requirements and can be configured by using the API `NVIC_SetPriority (IRQn, priority)` ⁽¹⁾ with the IRQ Number and the desired priority.

It is important to check how many priority levels are supported by the specific device from the Reference Manual or by checking the macro “`__NVIC_PRIO_BITS`” in the device header file (`<device>.h`). Furthermore there are some considerations for setting interrupt priorities listed below:

- In baremetal projects (No RTOS) there are no restrictions on configuring priorities as desired.
- For MQX RTOS projects, the scheduler limits the usage of interrupt priorities. The priority levels should comply with the next conditions:
 - 1) Priority level should be an even number.
 - 2) Priority level should be equal or higher than 2 times the value of `MQX_HARDWARE_INTERRUPT_LEVEL_MAX` value input in `mqx_init` structure if you want to use the MQX RTOS services in the interrupt service handler.

3. KSDK INTERRUPT HANDLING


This chapter contains a detailed explanation about interrupt handling with KSDK platform, when used for baremetal projects and MQX RTOS based projects.

3.1 Baremetal interrupt handling

In baremetal mode the application has full control of the interrupts logic by defining ISR functions or using the ones provided by the Peripheral Drivers.

3.1.1 Peripheral Driver interrupts

With KSDK Peripheral Drivers we have three separate functions involved with interrupt handling:



ISR entry functions	<ul style="list-style-type: none"> - These functions have names of the type “<peripheral>_IRQHandler” (e.g. I2CO_IRQHandler) or “<peripheral>_irq_IRQHandler” (e.g. UART0_RX_TX_IRQHandler). These are the valid entry points of the interrupt vector table. The names of these functions match those declared in the CMSIS startup assembly file (startup_<device>.S). - The weak handlers defined in CMSIS startup file are replaced by these functions. - ISR entry functions are defined in the files named “fsl_<peripheral>_irq.c”, which can be found in the particular driver folders in KSDK installation.
Driver IRQ functions	<ul style="list-style-type: none"> - These functions are driver defined and named “<peripheral>_DRV_IRQHandler”. They are invoked by the ISR entry functions. The purpose of Driver IRQ functions is to take action for the triggered interrupt based on common use cases. - Driver IRQ functions are defined in the driver implementation source files, which are part of the KSDK platform library build process.
Callbacks	<ul style="list-style-type: none"> - Some Peripheral Drivers (e.g. UART driver) allow the installation of callback functions. These functions may be called either by the ISR entry functions or the Driver IRQ functions when determined events occur. Callbacks are intended to execute user defined code apart from the common actions taken by the Driver IRQ routine. - Callbacks are installed by Driver APIs of the type “<peripheral>_DRV_InstallCallback()”.

IMPORTANT NOTE

The “`fsl_<peripheral>_irq.c`” files are not part of KSDK platform library build, so these files have to be added and built together with the application project, except when using Processor Expert, which generates ISR entry functions.

3.1.2 Bypass Peripheral Driver interrupt handlers

Although not a common practice, the user may want to bypass the default interrupt routine code for a particular Peripheral Driver and define a custom handler, for example to create a custom high level driver. This can be accomplished in two ways:

- **At compile time:** Define the custom ISR function with the same name as the one in CMSIS startup code assembly file. The corresponding file “`fsl_<peripheral>_irq.c`” has to be removed from the project, otherwise the linker may throw a Multiple Definition Error.
- **At runtime:** If the vector table is relocated to RAM, the user can install a custom ISR by calling the Interrupt Manager API `INT_SYS_InstallHandler (irqNumber, handler)` with the corresponding IRQ number and the name of the new ISR function.

Freescale Semiconductor

3.2 MQX RTOS interrupt handling

The MQX RTOS kernel takes ownership of the hardware interrupts and exceptions by having its own interrupt handling mechanism. By default all interrupts (except for some of the ARM® core interrupts) are mapped to a kernel ISR which takes action so that the scheduler and task context are not affected. Ultimately the kernel ISR calls a user registered ISR or a default ISR. Therefore MQX RTOS kernel creates its own “virtual” or separate vector table which has an entry for each interrupt number.

3.2.1 MQX interrupt installing

MQX RTOS delivers a set of interrupt related functions and macros. Only the most relevant functions for the scope of this document are described.

In MQX RTOS there are two main APIs for the installation of interrupt handling functions:

- **`_int_install_isr (vector , isr_ptr , isr_data)`**: This API installs an application-defined, interrupt-specific ISR, which MQX RTOS calls when the interrupt occurs. The ISR is installed to MQX RTOS owned ISR table. **`isr_ptr`** is the name of the ISR to install. **`isr_data`** is an optional parameter which is passed to the ISR function when called.
- **`_int_install_kernel_isr(vector, isr_ptr)`**: Installs an ISR handler for the specified interrupt directly to the hardware vector table, hence bypassing MQX RTOS and replacing the kernel ISR. **`isr_ptr`** is the name of the ISR function to install. This MQX RTOS API is only valid when vector table is located in RAM.

IMPORTANT NOTE

Both functions expect a “vector” parameter. The passed parameter must be the IRQ number of the corresponding interrupt and NOT the vector number.

Freescale Semiconductor

3.2.2 MQX interrupt functions

The next table provides a description of the different functions involved with interrupt handling in MQX for KSDK:

_int_kernel_isr()	<ul style="list-style-type: none">- This is the default interrupt function installed to the hardware vector table. All interrupts are intercepted by this function before calling the application defined ISR entry. The exception to this is when an interrupt ISR is installed directly to the hardware vector table, bypassing MQX.
_int_default_isr()	<ul style="list-style-type: none">- This ISR function is called from the _int_kernel_isr() whenever an interrupt is triggered for which there is no registered ISR entry function. During initialization, MQX RTOS points all of the interrupts to this default ISR and the application is responsible for installing its own ISR entries.
Application ISR entry	<ul style="list-style-type: none">- This is an application defined ISR entry. This type of ISR needs to be installed with the MQX API _int_install_isr(), so it can be called by the kernel ISR. This entry replaces the _int_default_isr() for the specific interrupt number.- Typically, these ISR entry functions are named "MQX_<peripheral>_IRQHandler" (e.g. MQX_I2CO_IRQHandler).
Driver IRQ functions	<ul style="list-style-type: none">- Driver defined functions named "<peripheral>_DRV_IRQHandler". These functions are invoked by the ISR entries and are designed to take action for the triggered interrupt according to common driver use cases.- Driver IRQ functions are defined inside of Driver specific source files, which are part of the KSDK platform library build.
Callbacks	<ul style="list-style-type: none">- Callbacks are functions that get called either by an application ISR entry or from the Driver IRQ functions. Callbacks allow the application to take action outside of the Driver IRQ flow.- To install callbacks the drivers provide APIs of the type "<peripheral>_DRV_InstallCallback()".

IMPORTANT NOTES

1- Due to the MQX RTOS interrupt handling mechanism, the names of the ISR entry functions must be different than those in the CMSIS startup code file (**startup_<device>.S**), otherwise the kernel ISR is not called when the corresponding interrupt triggers.

2- A valid approach is to copy the corresponding “**fsl_<peripheral>_irq.c**” file to the application and renaming the ISR entry functions by adding “MQX” at the beginning (e.g. MQX_I2C0_IRQHandler).

3.3 Operating System Abstraction layer (OSA)

As the name indicates, the OSA provides a level of abstraction to ease portability between Real Time Operating Systems or baremetal projects.

The OSA layer provides the next function related to interrupt handling:

- **OSA_InstallIntHandler (IRQNumber , handler)**: This API installs an ISR function for the specified IRQ number. The second parameter is a pointer to the desired interrupt handler function. The action taken by this API depends on the specific OSA type, as explained next.

Baremetal OSA:

fsl_os_abstraction_bm -> OSA_InstallIntHandler(): Installs the ISR for the specified IRQ number by calling the API **INT_SYS_InstallHandler()** from the Interrupt Manager.

MQX RTOS OSA:

fsl_os_abstraction_mqx -> OSA_InstallIntHandler(): Registers the ISR to the MQX RTOS interrupt handler, by calling the API **_int_install_isr()** from MQX RTOS kernel.

Freescale Semiconductor

4. KDS AND PROCESSOR EXPERT CONSIDERATIONS

This chapter shows some instructions and points to consider when applying the concepts of the previous sections for interrupt handling, when working in Kinetis Design Studio with one of 4 types of projects:

- 1) Baremetal KSDK projects
- 2) Baremetal + Processor Expert KSDK projects
- 3) MQX for KSDK projects
- 4) MQX for KSDK + Processor Expert projects

4.1 KSDK baremetal

Copy or link driver's IRQ file

1) Navigate to C:\Freescale\KSDK_1.2.0\platform\drivers\src\<peripheral>. Drag and drop the **fsl_<peripheral>_irq.c** file to any of your project source folders in KDS. In the figure below the PIT file is dragged to a folder called "DRIVER_IRQ".

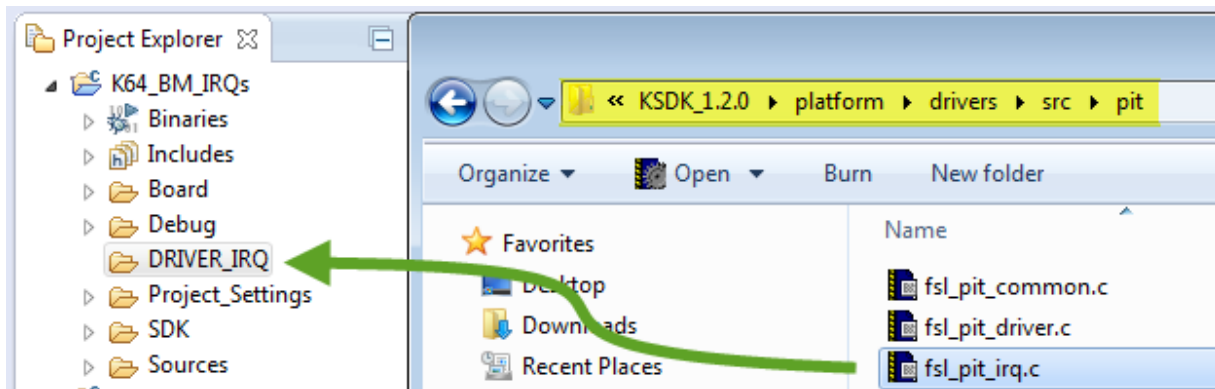


Figure 4.1 – Dragging IRQ file to project

2) KDS will ask if you want to link or copy the file to your project. The “Link” option creates a link to the original file in KSDK installation, while the “Copy” option creates a physical new copy of the file. Select according to your needs and click “OK”.

Freescal Semiconductor

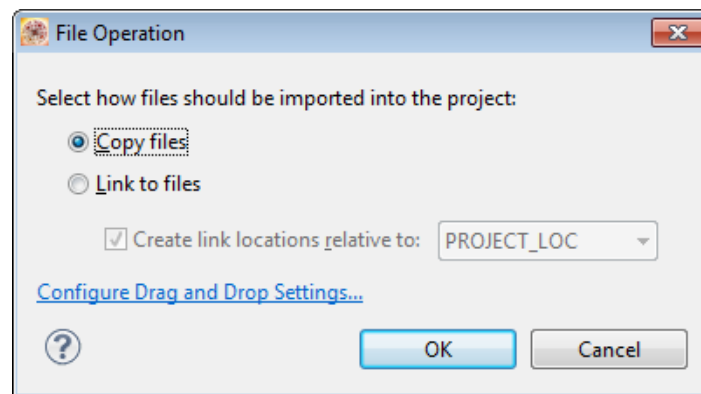


Figure 4.2 – Copy or link file

3) Finally KDS asks about adding new search paths. Just click on “Yes”.

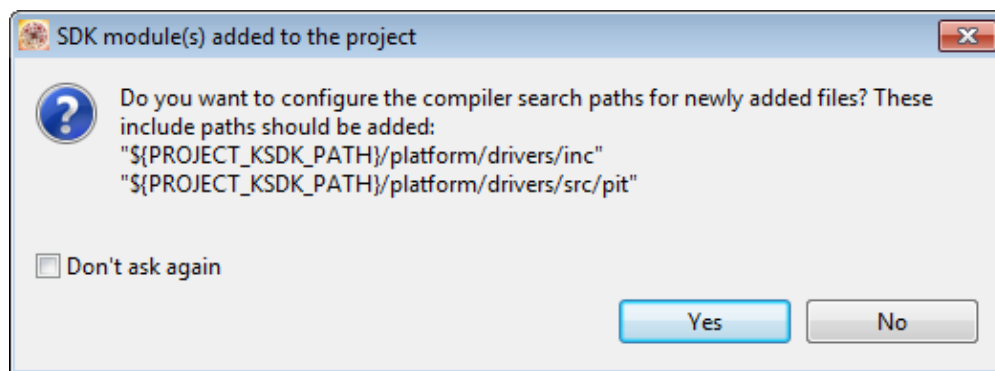


Figure 4.3 – Adding KSDK search paths

Using callbacks

The overall procedure to use driver callback functions involves 3 steps:

- 1) Define the callback function.
- 2) Initialize the Peripheral Driver.
- 3) Install callback with the corresponding driver API.

Example:

```
void lptmr_callback(void)
{
    // ... User defined code
}
```

Freescale Semiconductor

```
int main(void)
{
    lptmr_state_t lptmrState;

    lptmr_user_config_t lptmrUserConfig =
    {
        .timerMode = kLptmrTimerModeTimeCounter, /*! Use LPTMR in Time Counter mode */
        .freeRunningEnable = false, /*! When hit compare value, set counter back to zero */
        .prescalerEnable = false, /*! bypass prescaler */
        .prescalerClockSource = kClockLptmrSrcLpoClk, /*! use 1kHz Low Power Clock */
        .isInterruptEnabled = true
    };

    //...

    LPTMR_DRV_Init(LPTMR_INSTANCE, &lptmrState, &lptmrUserConfig);

    LPTMR_DRV_SetTimerPeriodUs(LPTMR_INSTANCE, 1000000);

    LPTMR_DRV_InstallCallback(LPTMR_INSTANCE, lptmr_isr_callback);

    LPTMR_DRV_Start(LPTMR_INSTANCE);
}
```

Relocating vectors to RAM

By default in a KSDK project (without Processor Expert) the vector table is not copied from Flash to RAM. The user can instruct the startup code to copy vectors to RAM by adding a linker symbol.

1) From KDS go to Project -> Properties -> C/C++ Build -> Settings -> Cross ARM C++ Linker -> Miscellaneous. In "Other linker flags" add the options: **-Xlinker --defsym=__ram_vector_table__=1**

Freescale Semiconductor

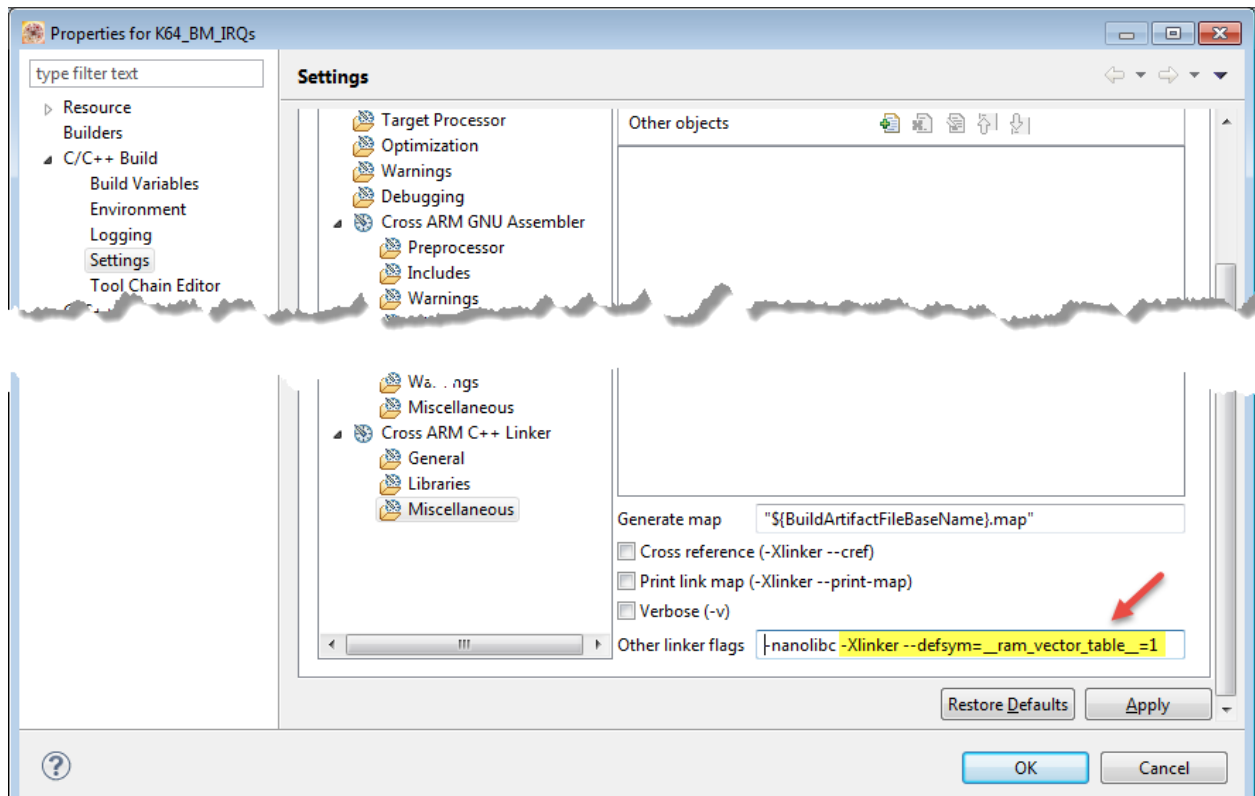


Figure 4.4 – Linker options to relocate vector table to RAM

2- Clean and build the project. Vectors should be copied to RAM during startup.

Freescale Semiconductor

4.2 KSDK baremetal + Processor Expert

Generating ISR entry functions

With Processor Expert there is no need to copy or link any peripheral IRQ file from KSDK installation, since Processor Expert generates all the required ISR entry functions automatically.

Depending on the specific peripheral component, if the IRQ handler is listed under the **Events** configuration tab, then the ISR entry is generated in the file “**Events.c**”, so it can be modified by the user. Otherwise only callback functions will be available in “**Events.c**” to add custom code, as long as the callback is enabled in the **Properties** tab of the component.

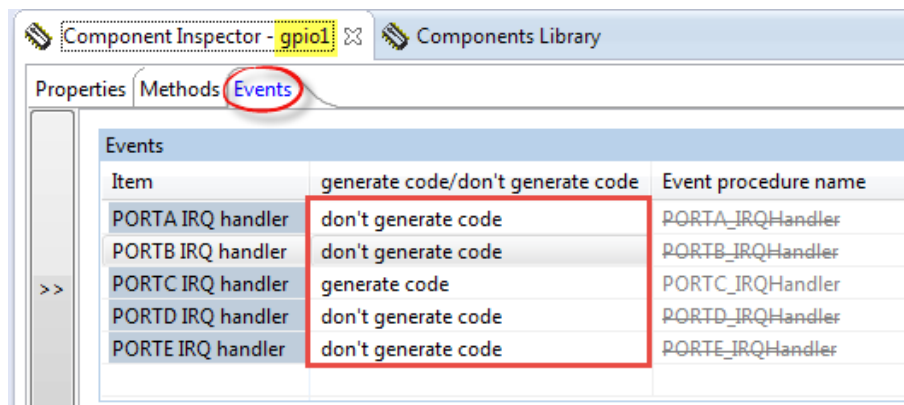


Figure 4.5 – IRQ handlers in the *Events* tab of “fsl_gpio” component

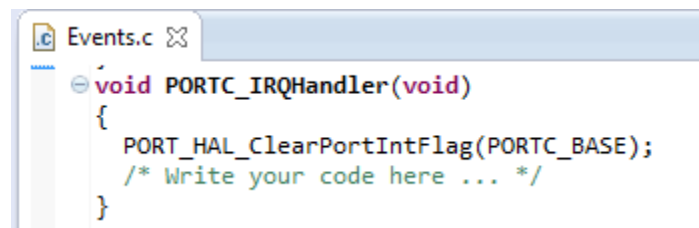


Figure 4.6 – ISR entry generated in “Events.c”

Freescale Semiconductor

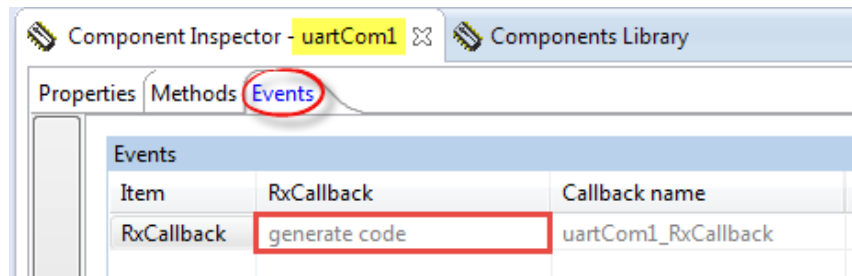


Figure 4.7 – Callback function in the *Events* tab of “fsl_uart” component

```

Events.c
void uartCom1_RxCallback(uint32_t instance, void * param)
{
    /* Write your code here ... */
}
    
```

Figure 4.8 – Callback function generated in “Events.c”

Installing callbacks

For the peripheral components supporting callbacks, there is one or more checkboxes under the **Properties** configuration tab to enable/disable callbacks, as well as custom name fields. The settings are reflected in the **Events** tab of the component and therefore in the generated code.

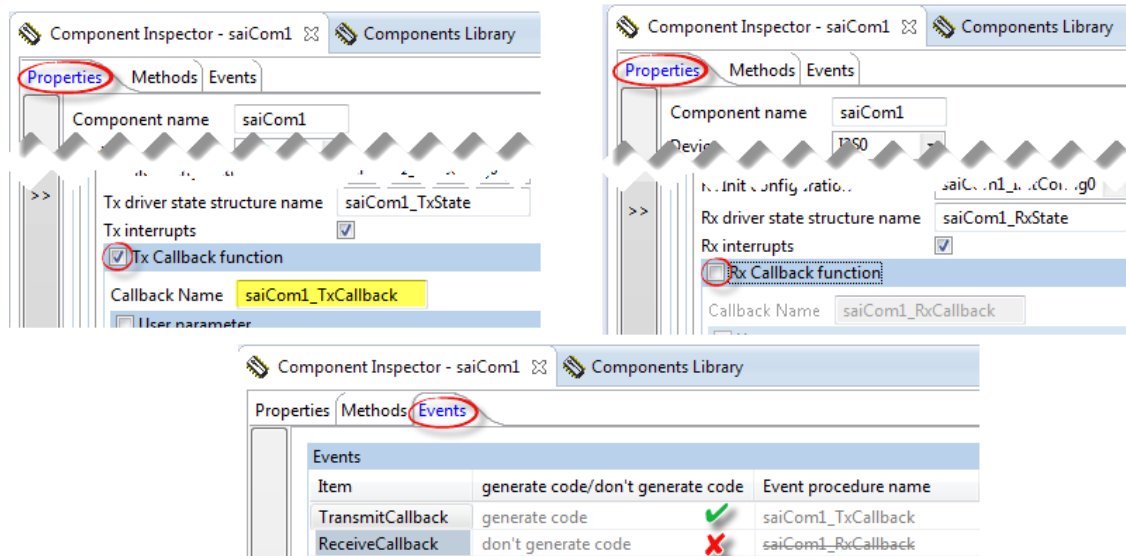


Figure 4.9 – Enabling/disabling callback functions. Pictures are for “fsl_sai” component

Freescale Semiconductor

Avoiding vector table relocation to RAM

By default in KSDK + Processor Expert projects, the generated linker file causes the vector table to be relocated to RAM. Sometimes to save space in RAM memory it is a good idea not to copy the interrupt vector table to RAM, especially if the application does not require dynamically installing new ISRs or updating the vector table.

- 1) From KDS in the Components view select the **Cpu** component and then in the Component Inspector view go to Build Options -> Generate linker file.

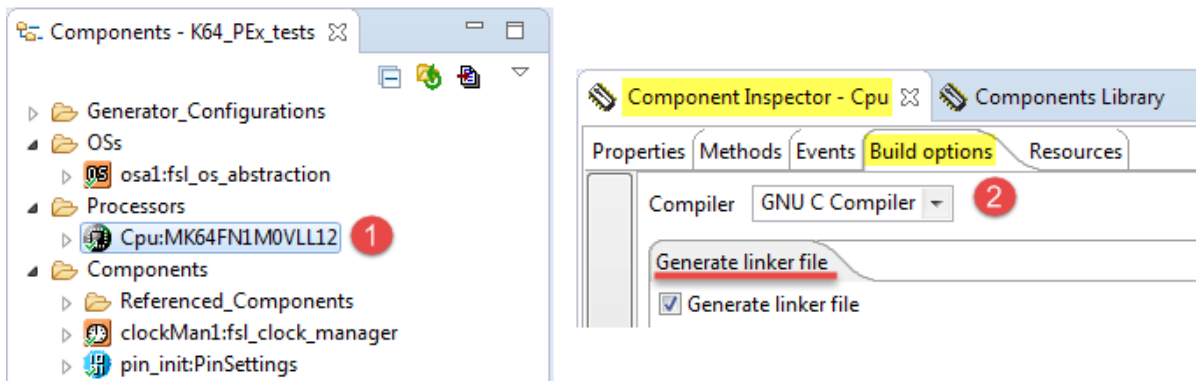


Figure 4.10 – CPU build options -> Generate linker file tab

- 2) Uncheck the option **Vector table copy in RAM**.

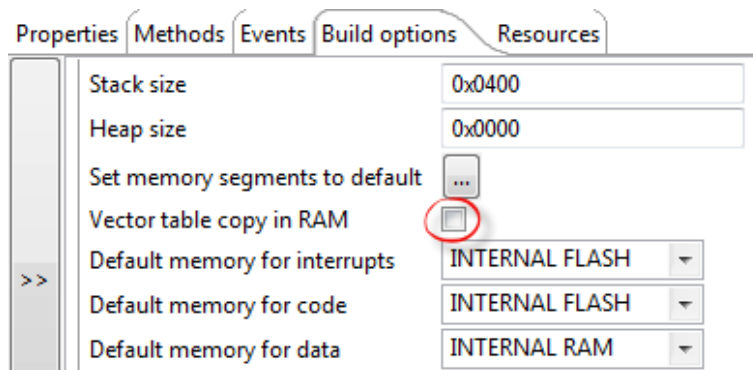


Figure 4.11 – Disabling vector table copy to RAM

Freescale Semiconductor

3) Processor Expert will ask if you want to adjust the memory ranges according to your new vector settings. Click on “Yes”

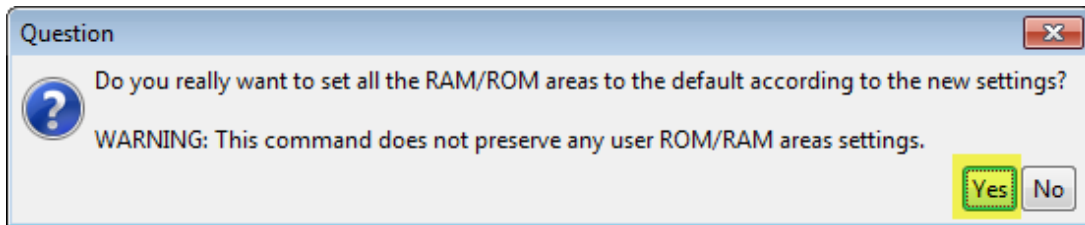


Figure 4.12 – Setting RAM/ROM memory ranges

NOTE

As indicated by the WARNING, user configured memory areas will not be preserved, so if you had custom memory segments or ranges (e.g. for a bootloader) make sure to reconfigure the segments as required.

The picture below shows an example of how disabling the vector copy in RAM would automatically modify the memory areas for a MK64FN1M0 device:

The first screenshot shows 5 memory areas. The second screenshot shows 4 memory areas, with a red circle around the number '4' in the header and a red box around the 'm_data' row. A large white arrow points from the first screenshot to the second.

#	ROM/RAM Area	Name	Qualifier	Address	Size
0	<input checked="" type="checkbox"/>	m_interrupts	RX	0x0	0x400
1	<input checked="" type="checkbox"/>	m_interrupts_ram	RW	0x1FFF0000	0x400
2	<input checked="" type="checkbox"/>	m_text	RX	0x410	0xFFBF0
3	<input checked="" type="checkbox"/>	m_data	RW	0x1FFF0400	0xFC00
4	<input checked="" type="checkbox"/>	m_data_2	RW	0x20000000	0x30000

#	ROM/RAM Area	Name	Qualifier	Address	Size
0	<input checked="" type="checkbox"/>	m_interrupts	RX	0x0	0x400
1	<input checked="" type="checkbox"/>	m_text	RX	0x410	0xFFBF0
2	<input checked="" type="checkbox"/>	m_data	RW	0x1FFF0000	0x10000
3	<input checked="" type="checkbox"/>	m_data_2	RW	0x20000000	0x30000

Figure 4.13 – Automatic changes in memory areas after disabling vectors copy

Freescal Semiconductor

4.3 MQX for KSDK

Copying and modifying IRQ files

The ISR entries in the KSDK IRQ files need to be modified so the names do not conflict with MQX.

- 1) Navigate to C:\Freescal\KSDK_1.2.0\platform\drivers\src<peripheral>. Drag and drop the **fsl_<peripheral>_irq.c** file to a folder in your KDS project.

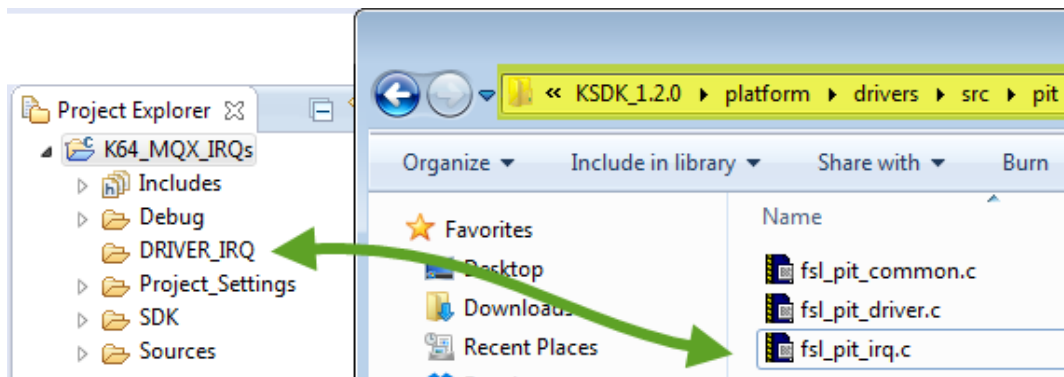


Figure 4.14 – Dragging peripheral IRQ file to MQX for KSDK project

- 2) A window will arise asking if you want to “Copy” or “Link” the file. Make sure to select the **Copy** option, so you can make proper changes and do not affect the original file.

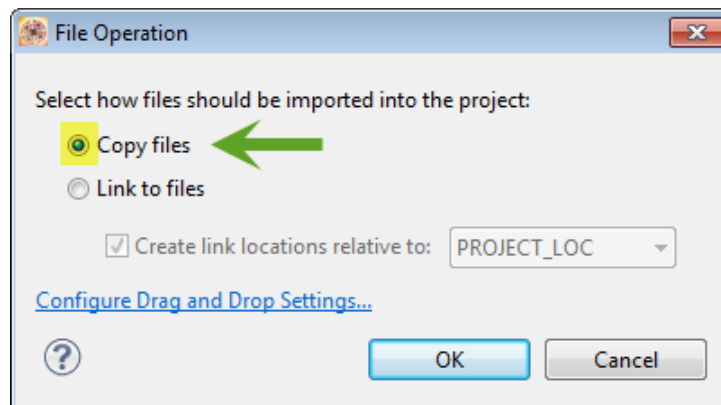


Figure 4.15 – Select “Copy files”

- 3) The last pop-up window will ask about adding search paths to the project. Just click on “Yes”.

Freescale Semiconductor

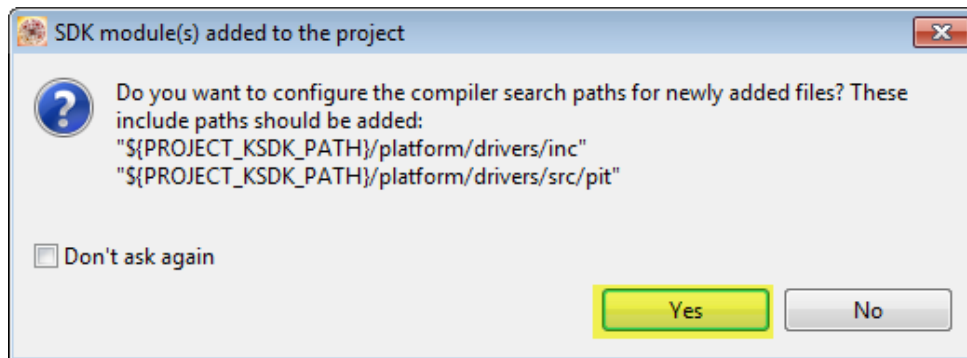


Figure 4.16 – Adding KSDK search paths

4) Open the IRQ file in KDS editor by double clicking on the file from the “Project Explorer”. You need to change the IRQ entry names by adding **MQX_** to each of the valid interrupts. Below an example for the file “**fsl_pit_irq.c**”:

```
void PIT0_IRQHandler(void)  →  void MQX_PIT0_IRQHandler(void)
{
  /* Clear interrupt flag.*/
  PIT_HAL_ClearIntFlag(g_pitBaseAddr[0], 0U);
  ...
}
```

Figure 4.17 – Adding “MQX_” prefix to peripheral ISR entry function name

Installing IRQ entries

Once the IRQ entry functions are defined as explained in the previous section, the user needs to call the function **OSA_InstallIntHandler** from the MQX application:

```
OSA_InstallIntHandler(PIT0_IRQn, MQX_PIT0_IRQHandler);
```

Using driver callbacks

To use driver callbacks in MQX the steps are the same as in baremetal projects. Please refer to section “[Using callbacks](#)” in **Chapter 4.1**.

Freescal Semiconductor

Installing vectors in RAM

With MQX for KSDK applications, the procedure to install the interrupt vector table in RAM is the same as for Baremetal applications. Please refer to Section “[Relocating vectors to RAM](#)” in **Chapter 4.1**.

4.4 MQX for KSDK + Processor Expert

Installing interrupts

MQX for KSDK requires the driver’s interrupts to be installed. When working with MQX for KSDK and Processor Expert, there are two critical steps required:

- 1- Check the box for “Install interrupt”.
- 2- Make sure the IRQ entry names have the **MQX_** prefix.

1) From every peripheral component added to the project, make sure that the “Install Interrupt” check box is marked. Below an example for **fsl_i2c** component:

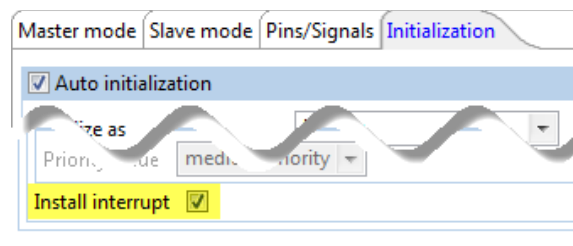


Figure 4.18 – Marking the “Install interrupt” checkbox

2) Make sure that the IRQ entry name is not the same as the one in startup assembly file, by adding the **MQX_** prefix to the name. For some peripheral components you will find such names in the **Properties** tab together with the “Install Interrupt” checkbox, while in other you will find the name under the **Events** tab. See the next examples:

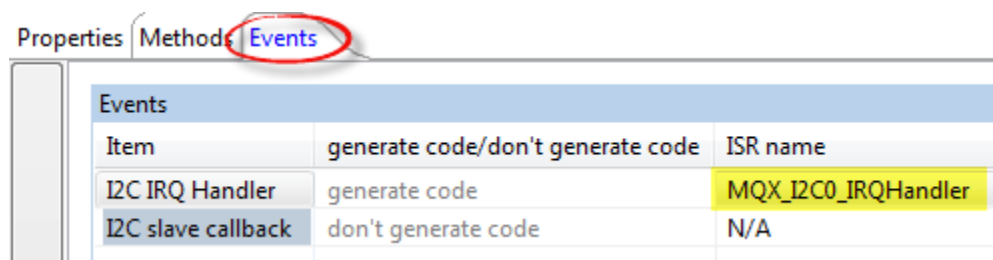


Figure 4.19 – IRQ entry name in the Events tab. This case is for “fsl_i2c” component

Freescale Semiconductor

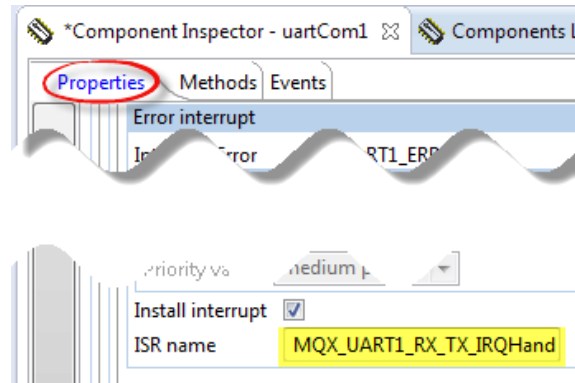


Figure 4.20 – IRQ entry name in the *Properties* tab. This case is for “fsl_uart” component

Using driver callbacks

The steps to install and use callbacks in a MQX for KSDK project with Processor Expert is the same as explained in section “[Installing callbacks](#)” in **Chapter 4.2**.

Freescal Semiconductor

5. REFERENCES

- 3- KSDK webpage: www.freescale.com/ksdk
- 4- KSDK documents: C:\Freescal\KSDK_1.2.0\doc
- 5- KSDK demo and example projects: C:\Freescal\KSDK_1.2.0\examples
- 6- MQX for KSDK RTOS documents: C:\Freescal\KSDK_1.2.0\doc\rtos\mqx
- 7- MQX for KSDK demo applications: C:\Freescal\KSDK_1.2.0\rtos\mqx\mqx\examples

- 8- How to create a baremetal KSDK project in KDS:
<https://community.freescale.com/docs/DOC-103288>

- 9- How to create a baremetal + PEx KSDK project in KDS:
<https://community.freescale.com/docs/DOC-104318>

- 10- How to create a MQX for KSDK project in KDS:
<https://community.freescale.com/docs/DOC-103405>

- 11- How to create a MQX for KSDK + PEx project in KDS:
<https://community.freescale.com/docs/DOC-103429>