MQX™ Lite Real-Time Operating System User Guide

Document Number: MQXLITEUG
Rev 1.1, 02/2014

freescale™

Contents

Section number **Title** **Page**

Chapter 1
Introduction

Chapter 2
Integration with Processor Expert

Chapter 3
Creating MQX Lite Application

Chapter 1
Introduction

MQX Lite is a version of the MQX™ Real-Time Operating System (RTOS) kernel targeted specifically for Microcontrollers (MCUs) either with limited RAM or limited FLASH memory size. Because this product is not a standalone package, such as the MQX RTOS, it is integrated into the Processor Expert (PEx) technology as a PEx component.

This document describes main features, integration into the PEx technology, and how to use MQX Lite to create custom applications.

Supplement this document with:

- Freescale MQX™ Lite RTOS Reference Manual - contains alphabetically-ordered listings of MQX Lite function prototypes and MQX Lite data types. This document is part of the PEx installation.
- Freescale MQX™ Real-Time Operating System User's Guide - contains detailed description of MQX components that are also used in the MQX Lite version. This document is part of the standard MQX RTOS installation.

This chapter gives an Overview of MQX Lite and describes MQX Lite Components.

1.1 Overview of MQX Lite

MQX Lite is an RTOS that manages the time of a microprocessor or microcontroller and allows:

- Multi-tasking
- Scheduling tasks with priorities
- Synchronization of the resource access
- Inter-task communication
- Interrupt handling

MQX Lite provides a run-time library of functions designed for real-time multitasking applications.

MQX consists of core (non-optional) and optional components. Functions, which are called either by MQX or an application, are the only functions included in the application image for core components. To match application requirements, an application can be extended by adding optional components.

Because MQX Lite is integrated with Processor Expert installation, it is easily configurable through GUI, which is the dedicated MQX Lite PEx component. The integration to PEx technology allows the usage of a wide range of Logical Device Drivers (LDD) that can be generated for different modules and different devices.

MQX Lite integration with PEx also allows an easy migration of the user application code across different Freescale platforms. There is no need for a Board Support Package (BSP) which is necessary for a standard MQX version. Instead, the set of PEx components and LDDs can be used to generate source code similar to the classic MQX BSP code. The generated code contains board-specific low-level startup code, processor, and board initialization code.

The user runs the PEx and creates a new bareboard project by using the New Project Wizard. This adds the MQX Lite PEx component into the created project. Adding MQX Lite into an existing project is also possible. Both ways of creating an MQX Lite application are described in this document. PEx GUI allows configuring the MQX Lite RTOS and defining application tasks and their parameters. When all required PEx components are added into the project and the entire configuration is complete, the source code for the selected compiler is generated. You can write the application by implementing already defined MQX Lite tasks, build, run and debug the application.

1.1.1 Comparison Between MQX Lite and MQX

These are the main differences between MQX Lite and standard MQX RTOS:

- MQX Lite does not use dynamic memory allocation. All kernel resources are allocated statically. However, the user application can still use the dynamic memory allocation offered by the optional lightweight memory allocation component.
- MQX Lite only supports priority based pre-emptive scheduler.
- MQX Lite does not allow dynamic task creation, all tasks resources are pre-allocated at compile time.
- MQX Lite supports "lightweight" subset of the MQX components: LW semaphore, LW event, Mutex, LW message, LW timer, and LW memory allocator.
- MQX Lite includes standard MQX interrupt handling.
- MQX Lite does not include a BSP nor any peripheral drivers. Instead, the PEx LDD drivers can be used in the end application.

These are the results of MQX Lite features and resource limitations:

- RTCS (Real-Time Communication Suite) is not supported in MQX Lite.
- MFS (Freescale MQX™ MFS™ Embedded File System) is not supported in MQX Lite.
- USB functionality is available using Freescale bare-metal USB stack (also a PEx component).
- MQX Shell library is not supported.

1.2 MQX Lite Components

The following table summarizes MQX Lite core and optional components. Each component is described in terms of MQX Lite specifics and differences when compared to the standard MQX.

Table 1-1. MQX Components

MQX Lite Component	*Include*	*Type*	*Function Prefix*
Initialization	Initialization and automatic task creation.	Core	`_mqxlite_`
Task management	Task creation, management, and termination.	Core	`_task_`
Scheduling	FIFO (also called priority-based preemptive) scheduling.	Core	`_sched_`
Lightweight semaphores	LW Semaphore synchronization mechanism.	Core	`_lwsem_`
Lightweight events	LW Events synchronization mechanism.	Optional	`_lwevent_`
Mutex	Mutual exclusion synchronization mechanism.	Optional	`_mutatr_, _mutex_`
Lightweight message queue	Inter-task communication.	Optional	`_lwmsgq_`
Interrupt and exception handling	Servicing all hw interrupts.	Core	`_int_`
Lightweight timer	Mechanism for calling application functions at periodic intervals.	Optional	`_lwtimer_`
Kernel log	MQX Lite activity recording.	Optional	`_klog_`
Lightweight memory allocation	Dynamic memory allocation	Optional	`_lwmem_`

1.2.1 Initialization

Initialization is a core component. It differs from the initialization component of the standard MQX. The `_mqx()` function of the standard MQX is split into two functions in MQXLite:

- `_mqxlite_init()` - This function is called during the Processor Expert internal initialization. Based on the values in MQX Lite initialization structure that is passed as `_mqxlite()` function argument, MQX Lite does the following:
  - It sets up and initializes the data that MQX Lite uses internally, including MQX Lite kernel data structure, ready queues, and the interrupt stack.
  - It creates lightweight semaphore for internal synchronization needs.
  - It starts the tick timer.
- `_mqxlite()` - This function starts MQX Lite and should be called from the user application, typically in `main()`, after the Processor Expert internal initialization code (`PE_low_level_init()` function). The `_mqxlite()` function:
  - Creates the idle task, which is active if no other task is ready.
  - Creates tasks that the task template list defines as autostart tasks.
  - Starts the task scheduler.

1.2.1.1 MQX Lite Initialization Structure

The MQX Lite initialization structure defines parameters of the application and target hardware.

```
typedef struct mqxlite_initialization_struct
{
    _mqx_uint           PROCESSOR_NUMBER;
    pointer             START_OF_KERNEL_MEMORY;
    pointer             END_OF_KERNEL_MEMORY;
    _mqx_uint           MQX_HARDWARE_INTERRUPT_LEVEL_MAX;
    _mem_size           INTERRUPT_STACK_SIZE;
    pointer             INTERRUPT_STACK_LOCATION;
    _mem_size           IDLE_TASK_STACK_SIZE;
    pointer             IDLE_TASK_STACK_LOCATION;
    TASK_TEMPLATE_STRUCT_PTR    TASK_TEMPLATE_LIST;
} MQXLITE_INITIALIZATION_STRUCT, * MQXLITE_INITIALIZATION_STRUCT_PTR;
```

For a description of each field, see the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

The MQX Lite initialization structure is generated automatically by the PEx based on the MQX Lite component settings and can be found in the `Generated_Code\<MQXLiteComponentName>.c` (for example, `MQX1.c`) file. This is an example of the initialization structure:

```
/* MQX Lite initialization structure */
const MQXLITE_INITIALIZATION_STRUCT  MQX_init_struct =
{
/* PROCESSOR NUMBER                 */  1,
/* START OF KERNEL MEMORY           */  __KERNEL_DATA_START,
/* END OF KERNEL MEMORY             */  __KERNEL_DATA_END,
/* MQX_HARDWARE_INTERRUPT_LEVEL_MAX */  1,
/* INTERRUPT STACK SIZE             */  sizeof(mqx_interrupt_stack),
/* INTERRUPT STACK LOCATION         */  mqx_interrupt_stack,
/* IDLE_TASK STACK SIZE             */  sizeof(mqx_idle_task_stack),
/* IDLE_TASK STACK_LOCATION         */  mqx_idle_task_stack,
/* TASK TEMPLATE LIST               */  (TASK_TEMPLATE_STRUCT_PTR)&MQX_template_list[0]
};
```

1.2.1.2 Task Template List

The task template list, which is a list of task templates (`TASK_TEMPLATE_STRUCT`), defines an initial set of templates. Tasks can be created on the processor by using this list. At initialization, MQX Lite creates one instance of each task, whose template defines it as an autostart task. In addition, while an application is running, it can create other tasks using a task template that the task template list defines. Dynamic task creation is not allowed in MQX Lite. All task resources must be pre-allocated statically at compile time. The end of the task template list is marked by a zero-filled task template.

```
typedef struct task_template_struct
{
    _mqx_uint           TASK_TEMPLATE_INDEX;
    TASK_FPTR           TASK_ADDRESS;
    _mem_size           TASK_STACKSIZE;
    _mqx_uint           TASK_PRIORITY;
    char               *TASK_NAME;
    _mqx_uint           TASK_ATTRIBUTES;
    uint32_t            CREATION_PARAMETER;
} TASK_TEMPLATE_STRUCT, * TASK_TEMPLATE_STRUCT_PTR;
```

```
/* MQX task template list */
const TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
  /* Task: Task1 */
  {
  /* Task number                    */  TASK1_TASK,
  /* Entry point                    */  (TASK_FPTR)Task1_task,
  /* Stack size                     */  TASK1_TASK_STACK_SIZE,
  /* Task priority                  */  9U,
  /* Task name                      */  "task1",
  /* Task attributes                */  (MQX_AUTO_START_TASK),
  /* Task parameter                 */  (uint32_t)(0)
  },
  /* Task: Task2 */
  {
  /* Task number                    */  TASK2_TASK,
  /* Entry point                    */  (TASK_FPTR)Task2_task,
  /* Stack size                     */  TASK2_TASK_STACK_SIZE,
  /* Task priority                  */  8U,
  /* Task name                      */  "task2",
  /* Task attributes                */  (0),
  /* Task parameter                 */  (uint32_t)(0)
  },
  TASK_TEMPLATE_LIST_END
};
```

1.2.2 Task Management

MQX Lite does not support dynamic task creation, which is one of main differences when compared to the Task Management component of the standard MQX. All tasks resources must be pre-allocated at compile time. The number of tasks is limited to one task activation at one time, either when MQX Lite starts, or at a designated moment in the user application. Usage of the `_task_create()` and `_task_create_blocked()` functions is disabled in MQX Lite. Instead, `_task_create_at()` function must be used.

Multiple tasks cannot be created from the same task template in MQX Lite, unless all tasks stacks are allocated statically and `_task_create_at()` function is correctly called in the application.

The application can dynamically change any task attribute. Only one task is active (has the processor) at any given time.

An exit function can be specified for each task, which MQX Lite calls when it terminates the task, and an exception handler, which MQX Lite calls when an exception occurs during task execution.

The list of MQX Lite Task Managements API functions (`_task_` prefix) is provided together with detailed description of each function in MQX Lite Reference Manual (MQXLITERM).

See the LED MQX Lite example code for the demonstration of the correct task creation. See MQX Lite Demo Applications for information about the location of MQX Lite examples.

1.2.3 Scheduler

MQX Lite manages the way tasks share the processor runtime (context switching) by using the Scheduler. Only one task is active (owns the processor) at any given time. The only scheduling policy that MQX Lite offers is FIFO, which is the priority based pre-emptive scheduler. The active task is the highest-priority task that is in "ready state". The active task runs until any of the following occurs:

- The active task voluntarily relinquishes the processor because it calls a blocking MQX Lite function.
- An interrupt occurs that has higher priority than the active task.
- A task that has priority higher than the active task becomes ready.

Scheduling is controlled by the task state and the position of the task in the ready queue. Each task is in one of the following logical states:

- **Blocked** - task is not ready to become active because it is waiting for a condition to occur; when the condition occurs, the task becomes ready.
- **Ready** - task is ready to become active, but it is not active because it is of the same priority as, or lower priority than, the active task.
- **Active** - task is running.
- **Terminated** - task has been destroyed/aborted.

Each task priority level has a ready queue. The active task is the first in the highest-priority ready queue. Tasks in each ready queue are in the FIFO order.

The list of MQX Lite Scheduler component API functions (`_sched_` prefix) is provided together with detailed description of each function in the MQX Lite Reference Manual (*MQXLITERM*).

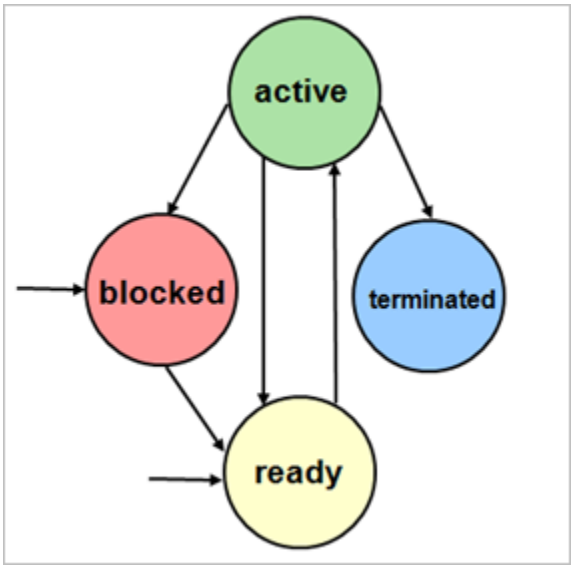

Figure 1-1. MQX Lite Task Logical States

1.2.4 Lightweight Semaphores

Lightweight semaphores support low-overhead synchronization of task accesses and shared resources. Lightweight semaphores require a minimal amount of memory and run quickly. They are used to:

- Control access to a shared resource (mutual exclusion).
- Signal the occurrence of an event.
- Allow two tasks to synchronize their activities.

There is no difference between lightweight semaphores in the standard MQX and the MQX Lite. The list of lightweight semaphore component API functions (`_lwsem_` prefix) is provided together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

See the LWSEM or LWDEMO MQX Lite example code for the demonstration of the correct Lightweight Semaphore usage. See MQX Lite Demo Applications for information about location of MQX Lite examples.

1.2.5 Lightweight Events

Lightweight events can be used either to synchronize two tasks or to synchronize a task and an ISR (Interrupt Service Routine). The lightweight event component consists of lightweight event groups, which are groupings of event bits. Any task can wait for event bits in a lightweight event group. If the event bits are not set, the task blocks. Any other

task or ISR can set the event bits. When the event bits are set, MQX puts all waiting tasks, whose waiting condition is met, into the task's ready queue. If the lightweight event group has autoclearing event bits, MQX Lite clears the event bits as soon as they are set and makes one task ready.

There is no difference between Lightweight Events in the standard MQX and the MQX Lite. This is the list of Lightweight Event component API functions:

(`_lwevent_` prefix) is stated together with detailed description of each function in the *Freesale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

See the LWEVENT or LWDEMO MQX Lite example code for the demonstration of the correct LW Event usage. See MQX Lite Demo Applications for information about the location of MQX Lite examples.

1.2.6 Mutex

A mutex provides mutual exclusion between tasks when they access a shared resource such as data or a device. Mutexes provide polling, FIFO queuing, priority queuing, spin-only and limited-spin queuing, priority inheritance, and priority protection. Mutexes are strict, which means that, unless a task had locked a mutex, it cannot unlock it.

The difference between the Mutex component in the standard MQX and the MQX Lite is minimal and involves statistical allocation of certain internal resources in MQX Lite. The list of Mutex component API functions (`_mutex_` , `_mutatr_` prefixes) is provided together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

See the MUTEX MQX Lite example code for the demonstration of the correct Mutex component usage. See MQX Lite Demo Applications for information about location of MQX Lite examples.

1.2.7 Lightweight Message Queue

The way that tasks communicate with each other is by exchanging messages. Lightweight message queues are a simpler, low-overhead, implementation of standard MQX messages. Tasks send messages to lightweight message queues and receive messages from lightweight message queues. A message in the message pool has a fixed size, which is a multiple of 32 bits. Blocking reads and blocking writes are provided.

To use a lightweight message queue, a task executes the following steps:

- Initializing a lightweight message queue. Message pool has to be allocated statically before the initialization of this component. When the task initializes the lightweight message queue, it also specifies the number of messages to be created and the size of each message.
- Sending messages to the lightweight message queue. If the queue is full, the task either blocks and waits, or fails sending. Task can be blocked after the message is sent.
- Receiving messages from the lightweight message queue. If the queue is empty, the reading task performs timeout. If the lightweight message queue is empty, the reading task can be blocked.

There is no difference between the Lightweight Message Queue in the standard MQX and the MQX Lite. The list of Lightweight Message Queue component API functions (`_lwmsgq_` prefix) is provided together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

See the LWDEMO MQX Lite example code for the demonstration of the correct Lightweight Message Queue component usage. See MQX Lite Demo Applications for information about location of MQX Lite examples.

1.2.8 Interrupts

MQX Lite handles all hardware interrupts as defined in the Interrupt Vector Table generated by PEx (see `vector.c`). Similarly to standard MQX, MQX Lite provides a first-level ISR (kernel ISR), which is written in assembly language. The first-level ISR runs before any other ISR, and performs these tasks:

- It saves the context of the active task.
- It switches to the interrupt stack.
- It calls the appropriate, second-level, (application) ISR.
- After the ISR has returned, it restores the context of the highest-priority ready task (context switch may occur).

First-level ISR can be replaced by the user-specific ISR on a per-vector basis.

MQX Lite provides second-level application ISRs that are coded as regular functions and are installed into into MQX Lite Interrupt Table through the Interrupt component API. Default second-level ISR for all possible interrupt sources is installed when MQX Lite starts. MQX Lite also supports the installation of the user-specific, default, second-level ISR. When a second-level ISR is called, the parameter, which the application defines when the application installs the ISR, is passed to the ISR.

Each entry of the MQX Lite Interrupt Table consists of:

- A pointer to the ISR to call.
- Data to pass as a parameter to the ISR.
- A pointer to an exception handler for that ISR.

MQX Lite supports installing an exception handler and enabling and disabling the Hardware interrupts.

The difference between the Interrupt component in the standard MQX and the MQX Lite is minimal and involves statistical allocation of certain internal resources. The list of Interrupt component API functions (`_int_` prefix) is stated together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

1.2.9 Lightweight Timer

Lightweight timer is an optional component that provides periodic notification to the application. An application creates a lightweight timer queue and adds timers to it. The timers expire at the same rate as the queue's period, but offset from the queue expiration time.

Lightweight timer is installed by creating a periodic queue, then adding a timer to expire at some offset from the start of the period. When adding a lightweight timer to the queue, a notification function is specified that is called by the MQX Lite tick ISR when the timer expires.

There is no difference between the Lightweight Timer component in the standard MQX and the MQX Lite. The list of Lightweight Timer component API functions (`_lwtimer_` prefix) is provided together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

1.2.10 Kernel Log

Kernel Log allows recording MQX Lite activity, which means that it is configurable to record all MQX Lite function calls, context switches, and interrupt servicing.

Because of the MQX Lite restriction in dynamic memory allocations, `_klog_create()` functions cannot be used in MQX Lite. However, the `_klog_create_at()` function, which creates the kernel log at the specified location (statically allocated), is available.

The list of Kernel Log component API functions (`_klog_` prefix) is provided together with detailed description of each function in the *Freescale MQX™ Lite RTOS Reference Manual* (MQXLITERM).

1.2.11 Lightweight Memory Allocation

To allow the use of the dynamic memory allocation in target application, MQX Lite provides Lightweight Memory Allocation component. Allocation process is similar to using standard `malloc()` and `free()` functions available in most C run-time libraries. The difference is that MQX Lightweight Memory Allocation component provides a task safe mechanism to alloc/free memory from concurrently running tasks. MQX Lightweight Memory Allocators are also supported by the MQX Task Aware Debugging plug-in providing detailed information about allocated memory.

MQX Lite allocates memory blocks from its default memory pool, which is mapped to a standard application heap. Tasks can also create memory pools outside the default memory pool and allocate memory blocks from there.

When MQX Lite allocates a memory block, it allocates a block which is the requested size or larger (the block might be slightly larger to avoid memory fragmentation).

By default the Lightweight Memory Allocation feature is turned off.

To enable its usage in an application the user should perform these steps:

- Set Lightweight Memory Allocation option to “yes” in MQX Lite component properties.
- Set a heap size of the default memory pool.
  - Right click on the CPU component -> Build options -> Heap size

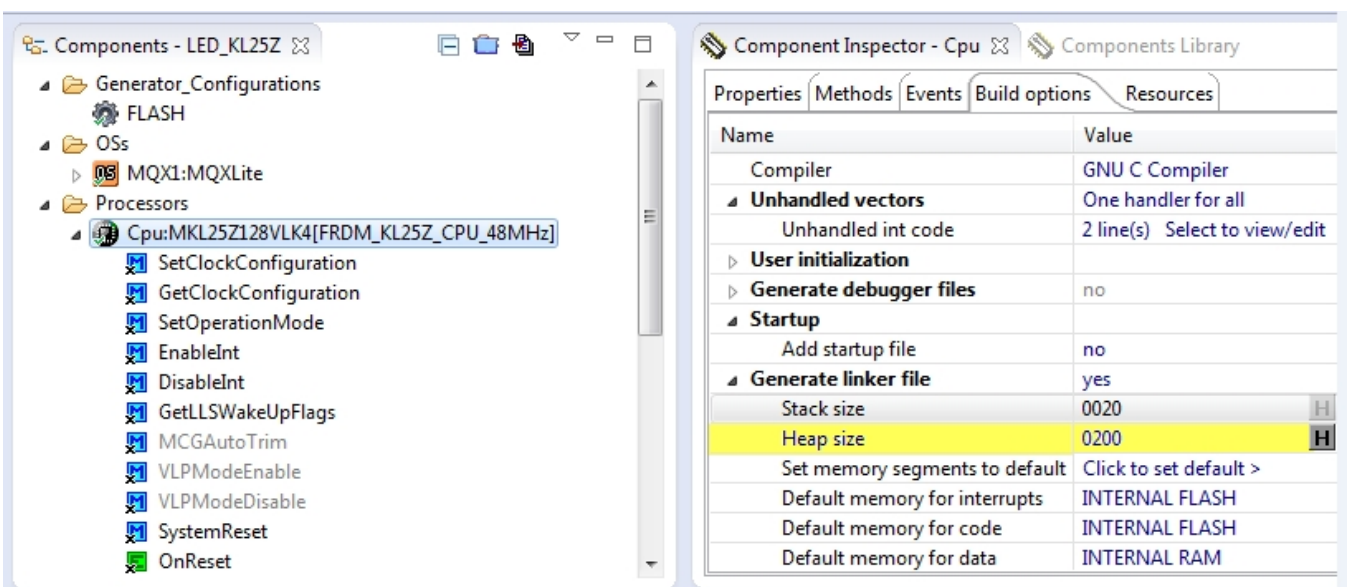

Figure 1-2. MQX Lite Heap Size Setting

Chapter 2
Integration with Processor Expert

MQX Lite is integrated with Processor Expert in both products: Microcontrollers Driver Suite V10.x and CodeWarrior V10.x.

This chapter describes:

2.1 MQX Lite Component Interface

The MQX Lite component is used to configure the MQX Lite core and define the interface between MQX Lite and Processor Expert.

Main features of MQX Lite component are:

- MQX Lite core configuration
- Interrupt service routines installation mechanism
- Vector table management

2.2 MQX Lite Generated Files

Processor Expert MQX Lite component generates following files to the project directory in a user Eclipse workspace:

- `Generated_Code\<MQXLiteComponentName>.c` and `.h` (for example, `MQX1.c`, `MQX1.h`) - Implementation of MQX Lite interface, time notification of MQX Lite scheduler,

definition of MQX Lite initialization structure, definition of task stacks, definition of `PEX_RTOS_INIT()` and `PEX_RTOS_START()` macros.

- `Generated_Code\task_template_list.c and .h` - definition of task template list based on MQX Lite component configuration.
- `Generated_Code\user_config.h` - MQX Lite configuration file based on MQX Lite component configuration.
- `Sources\mqx_task.c and .h` - Processor Expert user module containing implementation/ definition of task routines. The user can add custom code to this module and implement the body of a task function here. A content of this file is not regenerated by Processor Expert and all user changes are kept. When a new task is specified in the MQX Lite component properties tab, a default MQX Lite task routine is created. The user can modify an implementation of this function according to the application needs.

```
/*
** ===================================================================
**     Event       :  Task1_task (module mqx_tasks)
**
**     Component   :  Task1 [MQXLite_task]
**     Description :
**         MQX task routine. The routine is generated into mqx_tasks.c
**         file.
**     Parameters  :
**         NAME            - DESCRIPTION
**         task_init_data  -
**     Returns     : Nothing
** ===================================================================
*/
void Task1_task(uint32_t task_init_data)
{
  int counter = 0;

  while(1) {
    counter++;

    /* Write your code here ... */

  }
}
```

Figure 2-1. MQX Lite Task Routine

2.3 Vector Table Management

Interrupt vector table is generated by Processor Expert into the `Vectors.c file`. Because MQX Lite implements its own interrupt management, `_int_kernel_isr()` the interrupt routine is installed to the interrupt vector table by default. On Kinetis platforms with

ARM® Cortex®-M0+ code, the first fifteen interrupt vectors are not managed by MQX Lite and Processor Expert installs its own default <CpuComponentName> `_Interrupt()` interrupt service routine to these interrupt vectors.

```
__attribute__ ((section (".vectortable"))) const tVectorTable __vect_table = { /* Int

  /* ISR name                          No. Address      Pri Name
  &__SP_INIT,                       /* 0x00  0x00000000   -   ivINT_Initial_Stack_Pc
  {
  (tIsrFunc)&__boot,                /* 0x01  0x00000004   -   ivINT_Initial_Program_
  (tIsrFunc)&Cpu_INT_NMIInterrupt,  /* 0x02  0x00000008   -2   ivINT_NMI
  (tIsrFunc)&Cpu_Interrupt,         /* 0x03  0x0000000C   -1   ivINT_Hard_Fault
  (tIsrFunc)&Cpu_Interrupt,         /* 0x04  0x00000010   -   ivINT_Reserved4
  (tIsrFunc)&Cpu_Interrupt,         /* 0x05  0x00000014   -   ivINT_Reserved5
  (tIsrFunc)&Cpu_Interrupt,         /* 0x06  0x00000018   -   ivINT_Reserved6
  (tIsrFunc)&Cpu_Interrupt,         /* 0x07  0x0000001C   -   ivINT_Reserved7
  (tIsrFunc)&Cpu_Interrupt,         /* 0x08  0x00000020   -   ivINT_Reserved8
  (tIsrFunc)&Cpu_Interrupt,         /* 0x09  0x00000024   -   ivINT_Reserved9
  (tIsrFunc)&Cpu_Interrupt,         /* 0x0A  0x00000028   -   ivINT_Reserved10
  (tIsrFunc)&Cpu_Interrupt,         /* 0x0B  0x0000002C   -   ivINT_SVCall
  (tIsrFunc)&Cpu_Interrupt,         /* 0x0C  0x00000030   -   ivINT_Reserved12
  (tIsrFunc)&Cpu_Interrupt,         /* 0x0D  0x00000034   -   ivINT_Reserved13
  (tIsrFunc)&Cpu_Interrupt,         /* 0x0E  0x00000038   -   ivINT_PendableSrvReq
  (tIsrFunc)&_int_kernel_isr,       /* 0x0F  0x0000003C   2   ivINT_SysTick
  (tIsrFunc)&_int_kernel_isr,       /* 0x10  0x00000040   -   ivINT_DMA0
  (tIsrFunc)&_int_kernel_isr,       /* 0x11  0x00000044   -   ivINT_DMA1
  (tIsrFunc)&_int_kernel_isr,       /* 0x12  0x00000048   -   ivINT_DMA2
  (tIsrFunc)&_int_kernel_isr,       /* 0x13  0x0000004C   -   ivINT_DMA3
  (tIsrFunc)&_int_kernel_isr,       /* 0x14  0x00000050   -   ivINT_Reserved20
  (tIsrFunc)&_int_kernel_isr,       /* 0x15  0x00000054   -   ivINT_FTFA
  (tIsrFunc)&_int_kernel_isr,       /* 0x16  0x00000058   -   ivINT_LVD_LVW
  (tIsrFunc)&_int_kernel_isr,       /* 0x17  0x0000005C   -   ivINT_LLW
  (tIsrFunc)&_int_kernel_isr,       /* 0x18  0x00000060   -   ivINT_I2C0
```

Figure 2-2. Interrupt Vector Table

2.4 Interrupt Service Routines Installation Mechanism

When MQX Lite component is a part of the project, all Processor Expert components install their interrupt service routines through the MQX Lite interrupt installation mechanism.

2.5 MQX Lite Component Interface

Similarly to all other Processor Expert components, MQX Lite component also has its own graphical user interface. The following component properties and methods can be configured by using the Component Inspector view:

2.5.1 Properties

This section describes properties of the MQX Lite component. Properties are parameters of the component that influence the generated code. See the *Processor Expert User Guide* for more details.

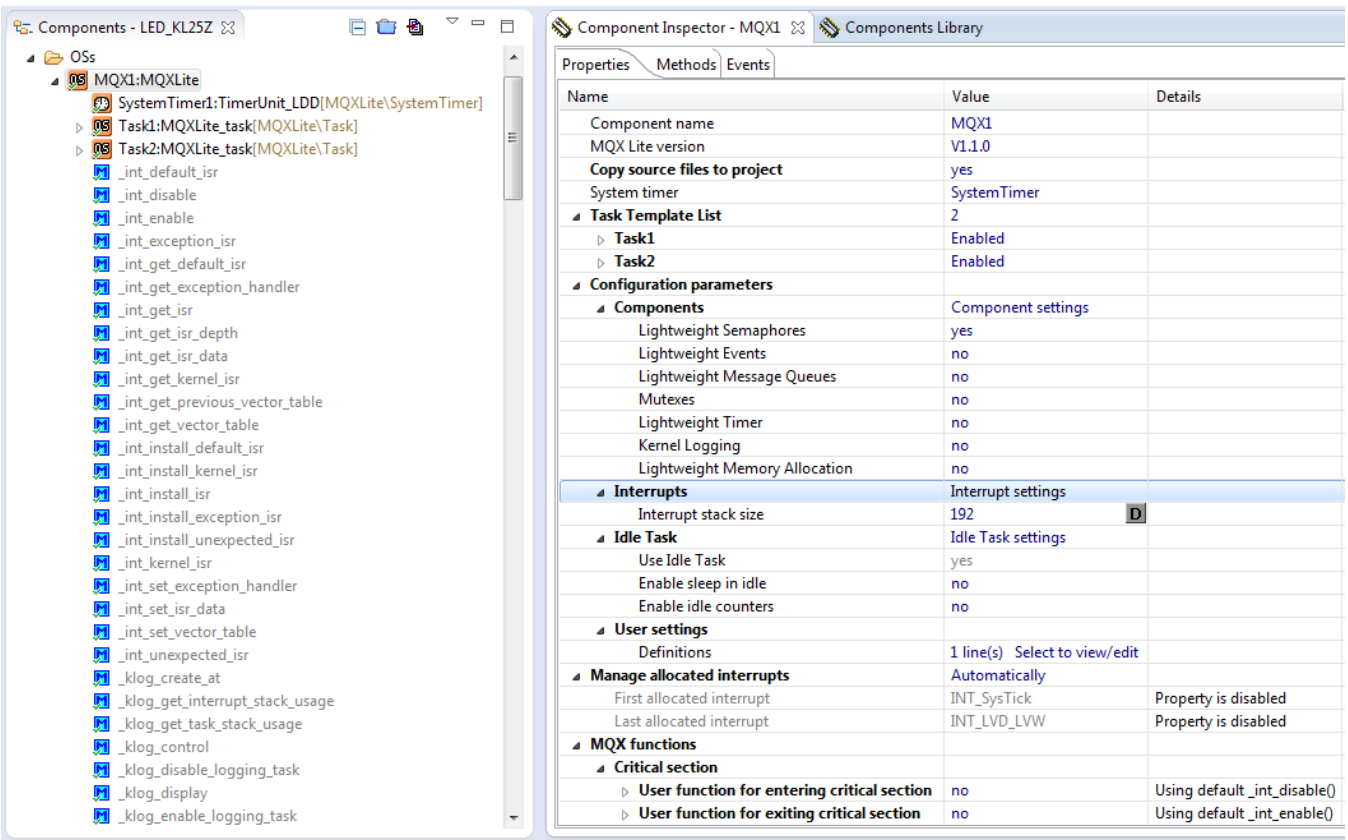

Figure 2-3. Properties of the MQX Lite Component

- **Component name** - Name of the component.
- **MQX Lite version** - Specifies which MQX Lite version is used. It affects which MQX Lite source files are copied into the project.
- **Copy source files to project** - Specifies whether MQX Lite source files are copied into a project after code generation.
- **System timer** - System timer is used for kernel notification (for details about settings see Component Inheritance & Component Sharing).
- **Task Template List** - The task template list is an array of task template structures terminated by a zero-filled element. The MQX Lite initialization structure contains a pointer to this list.
- **Configuration parameters** - MQX Lite configuration parameters.

- **Components** - MQX Lite components (LW Semaphores, Mutexes, LW Events, LW Message Queues, Logs, and Memory Allocation) settings.
  - **Lightweight Semaphores** - Specifies whether Lightweight Semaphores are allowed in an MQX Lite project. As LW Semaphore is a core required component disabling this option is not recommended.
  - **Lightweight Events** - This property specifies whether Lightweight Events can be used in an MQX Lite project.
  - **Lightweight Message Queues** - This property specifies whether Lightweight Message Queue can be used in an MQX Lite project.
  - **Mutexes** - This property specifies whether Mutexes can be used in an MQX Lite project.
  - **Lightweight Timer** - This property specifies whether Lightweight Timer can be used in an MQX Lite project.
  - **Kernel logging** - This property specifies whether Kernel Logging is allowed in an MQX Lite project.
  - **Lightweight Memory Allocation** - This property specifies whether Lightweight Memory Allocation is allowed in an MQX Lite project.
- **Interrupts** - Interrupt settings.
  - **Interrupt stack size** - Size of the stack space used by all ISRs.
- **Idle Task** - Idle Task settings.
  - **Use Idle Task** - When enabled, the kernel creates the idle task; otherwise, if a task is not ready, it does not run. The use of an idle task is mandatory for Kinetis platforms.
  - **Enable sleep in idle** - Enables the use of the WFI (wait for interrupt) instruction in the idle task loop.
- **User settings** - User settings.
  - **Definitions** - Other user defined values placed into the `user_config.h`.

- **Manage allocated interrupts** - This property specifies the range of used interrupt vectors. The range can be specified automatically or manually by user. This influences the size of the MQX Lite interrupt table.

  The following items are available only if the group is enabled (the value is "By user"):

  - **First allocated interrupt** - First user allocated interrupt vector can be chosen from the list.
  - **Last allocated interrupt** - Last user allocated interrupt vector can be chosen from the list.
- **MQX functions** - Contains the RTOS specific settings. The settings in this section are common for all components generally called RTOS Adaptors. The MQX Lite component is a special kind of RTOS Adaptor, which makes the code generated by all other PEx components compatible with MQX Lite.

- **Critical section** - Definition of the RTOS API which provides the critical section handling for HAL driver code.
  - **User function for entering critical section** - The open critical section function should introduce the code which cannot be interrupted by the ISRs. This function can be also called from ISRs and/or within a context where ISRs are already disabled. In this situation the function has no effect (except that the pair call to critical section close function is also ignored - see property "User function for exiting critical section"). The function has no parameters and returns no value.

    The following item is available only if the group is enabled (the value is "yes"):

  - **User function name** - The name of the user function (see above for description).
- **User function for exiting critical section** - The close critical section function should terminate the code which cannot be interrupted by the ISRs. This function should always have a pair call to the open critical section (see property "User function for entering critical section"). If the call to the pair open critical section was ignored, the respective close should be also ignored. The function has no parameters and returns no value.

  The following item is available only if the group is enabled (the value is "yes"):

  - **User function name** - The name of the user function (see above for description).

2.5.2 Methods

Component methods are functions/subroutines, which the user can call, intended for the component runtime control. See the *Processor Expert User Guide* for more details.

MQX Lite component methods are identical to MQX Lite API functions that are described in the *Freescale MQX™ Lite RTOS Reference Manual*.

2.5.3 Events

There are no events defined for the MQX Lite component.

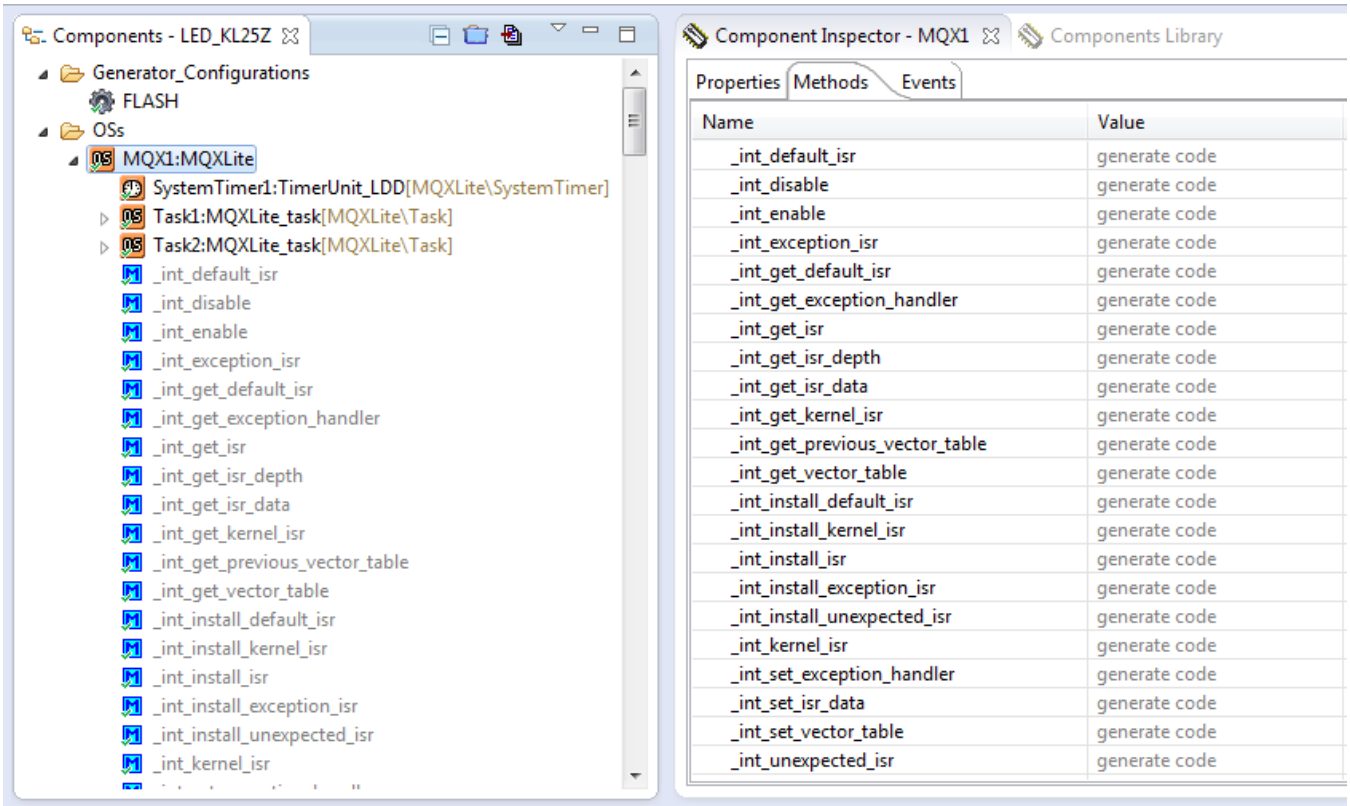

Figure 2-4. MQX Lite Component's Methods

2.6 MQXLite_task Component Interface

MQXLite_task components are dynamically created when changing the Task Template List property of the MQX Lite component. Based on the defined number of application tasks, the corresponding number of MQXLite_task components are created and assigned as inherited components (see chapter "Component Inheritance and Component Sharing" of the *Processor Expert User Guide*). Use the options in the Component Inspector of the particular MQXLite_task to set up the tasks.

2.6.1 Properties

This section describes properties of the `MQXLite_task` component. The MQX Lite task template list is generated based on the settings in this component.

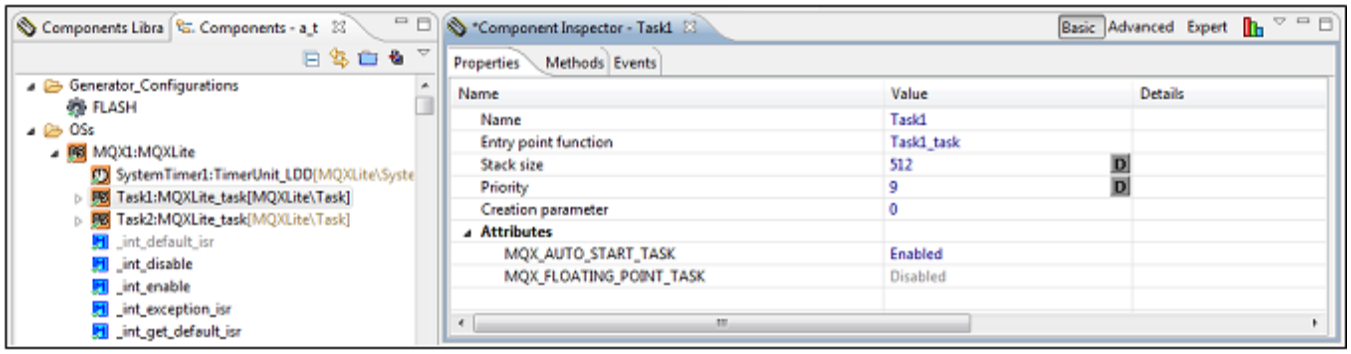

Figure 2-5. Properties of the MQXLite_task Component

- **Name** - String name for the task.
- **Entry point function** - Root function name for the task. This function is called when the task is created. It's up to the user to implement this function (in `mqx_task.h/.c` by default).
- **Stack size** - Size of the stack space required by the task.
- **Priority** - Software priority of the task. Priorities start at 0, which is the highest priority. 1, 2, 3, and so on, are progressively lower priorities. Due to mapping of task priority levels and interrupt levels on some microcontroller platforms, it is recommended to start assigning the priorities at number 7 for the highest-priority task.
- **Creation parameter** - The value stored in this property is the default value that is passed as the creation parameter to the task when created.
- **Attributes** - Enables/disables task attributes relevant for the selected platform.
  - **MQX_AUTO_START_TASK** - When MQX Lite starts, it creates one instance of the task and makes it ready.
  - *Settings supported for Freescale ColdFire and ColdFire+ derivatives only.* **MQX_DSP_TASK** - MQX Lite saves the DSP coprocessor registers as part of the task's context. If the DSP registers are separate from the normal registers, MQX Lite manages their context independently during task switching. MQX Lite saves or restores the registers only when a new DSP task is scheduled to run.
  - **MQX_FLOATING_POINT_TASK** - Task uses the floating point co-processor. MQX Lite saves floating-point registers as part of the task's context.

2.6.2 Methods

There are no methods defined for the `MQXLite_task` component.

2.6.3 Events

This section describes `MQXLite_task` component's events. Events are call-back functions called when an important event occurs. Refer to the *Processor Expert User Guide* for more details.

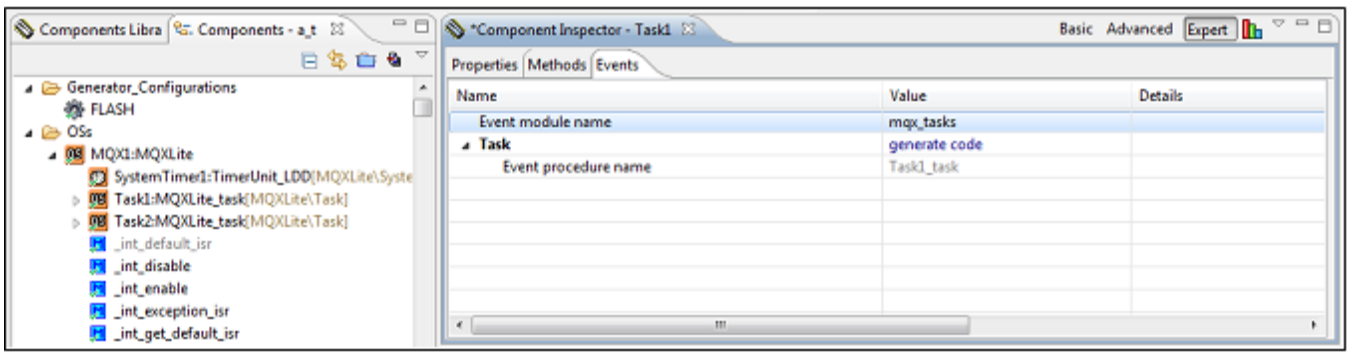

Figure 2-6. MQXLite_task Component's Properties

- **Event module name** - Name of the user module (without extension), where the task from this component is placed.
- **Task** - MQX Lite task routine. This routine is generated into `mqx_tasks.c` file and called when the task is created.

2.7 Initialization of MQX Lite RTOS in an Application

As described in the Initialization section, the MQX Lite initialization is divided into two steps:

Initialization and start of the MQX Lite core. The MQX Lite core is initialized in the `PE_low_level_init(void)` function defined in CPU component's module. The `PEX_RTOS_INIT()`macro is used to call `_mqxlite_init()`function.

```
/*
** ===================================================================
**     Method      :  PE_low_level_init (component MKL25Z128LK4)
**
**     Description :
**         Initializes beans and provides common register initialization.
**         The method is called automatically as a part of the
**         application initialization code.
**         This method is internal. It is used by Processor Expert only.
** ===================================================================
*/
void PE_low_level_init(void)
{
  #ifdef PEX_RTOS_INIT
    PEX_RTOS_INIT();                   /* Initialization of the selected RTOS.
  #endif
```

Figure 2-7. MQXLite Initialization

MQX Lite is stared in the main module. The macro `PEX_RTOS_START()` is used to call `_mqxlite()` function.

```
int main(void)
/*lint -restore Enable MISRA rule (6.3) checking. */
{
  /* Write your local variable definition here */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***

  /* Write your code here */
  /* For example: for(;;) { } */

  /*** Don't write any code pass this line, or it will be deleted during code
  /*** RTOS startup code. Macro PEX_RTOS_START is defined by the RTOS componen
  #ifdef PEX_RTOS_START
    PEX_RTOS_START();                  /* Startup of the selected RTOS. Macro
  #endif
  /*** End of RTOS startup code.  ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/
```

Figure 2-8. MQXLite Start

Chapter 3
Creating MQX Lite Application

This chapter describes:

3.1 Creating a New MQX Lite Application

The easiest way to create a new MQX Lite application is to use the MQX Lite Project Wizard. The following steps describe how to create a new MQX Lite application by using MQX Lite project wizard.

1. Open the MQX Lite Project Wizard and select **File -> New -> Processor Expert MQX Lite Project**, specify a Project name and click **Next**.

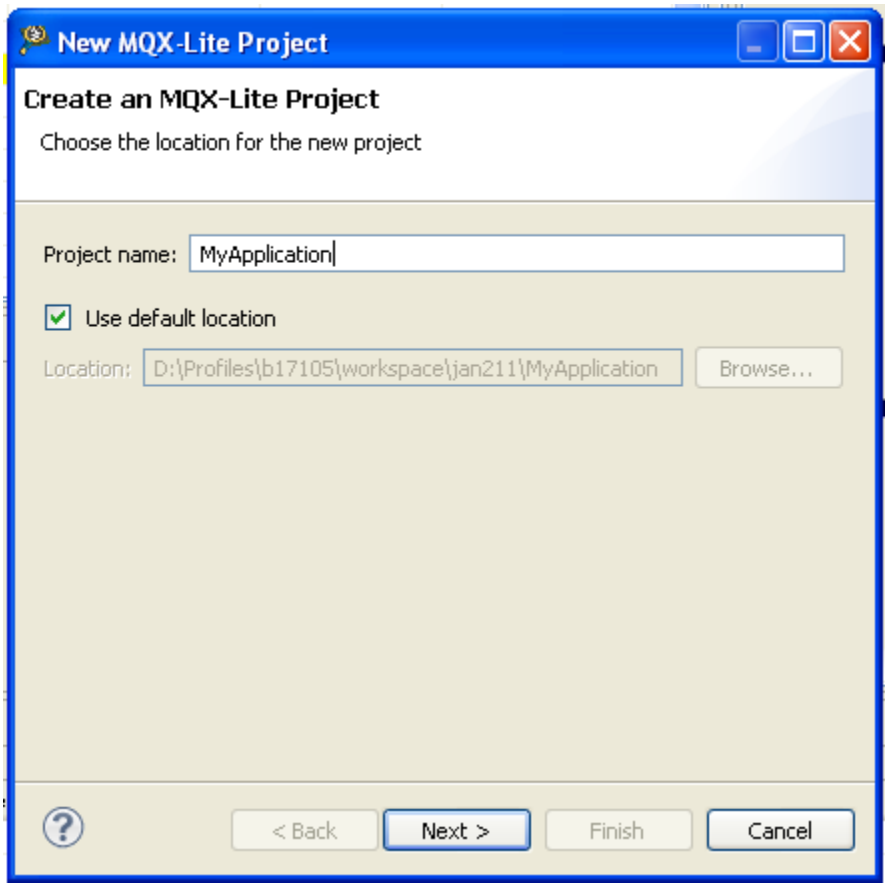

Figure 3-1. Create Processor Expert MQX Lite Project

2. Select a target derivative and click **Next**.

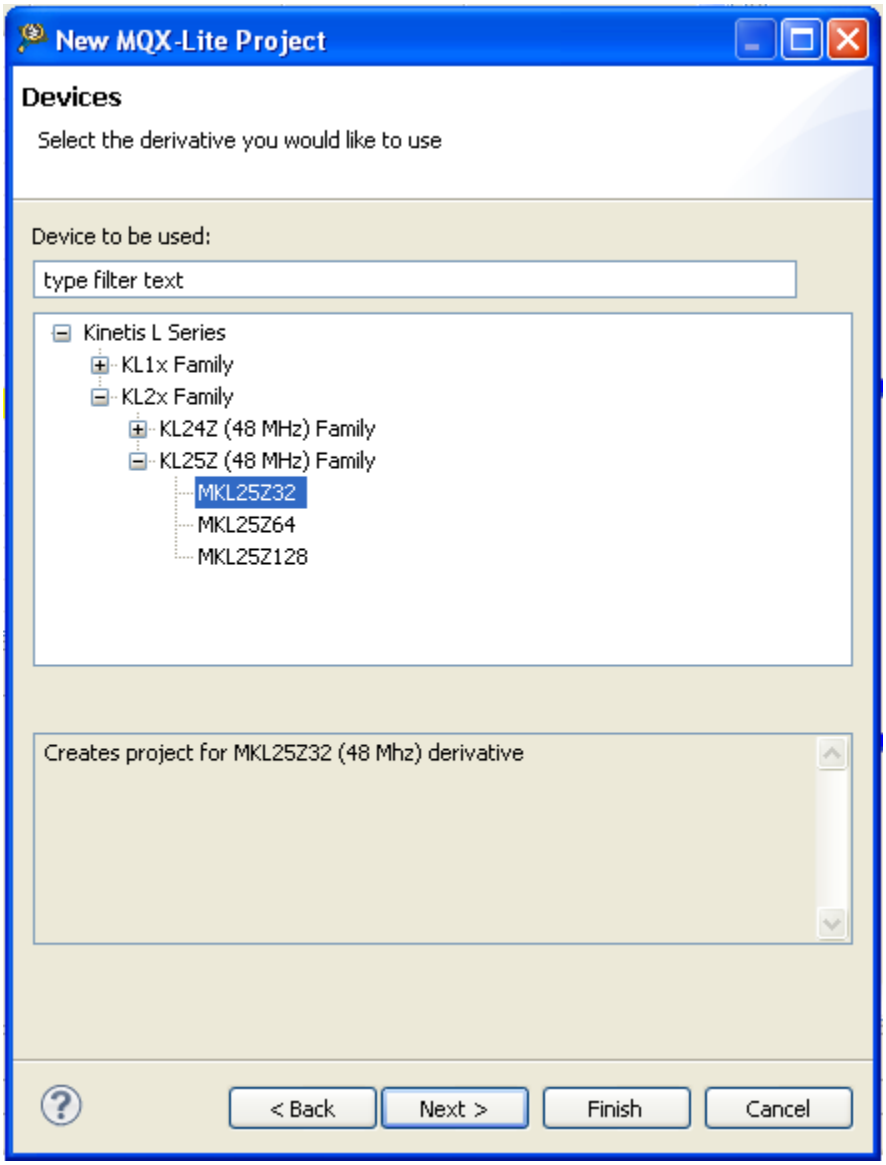

Figure 3-2. Devices Selection Page

3. Select the default connections and click **Next**.

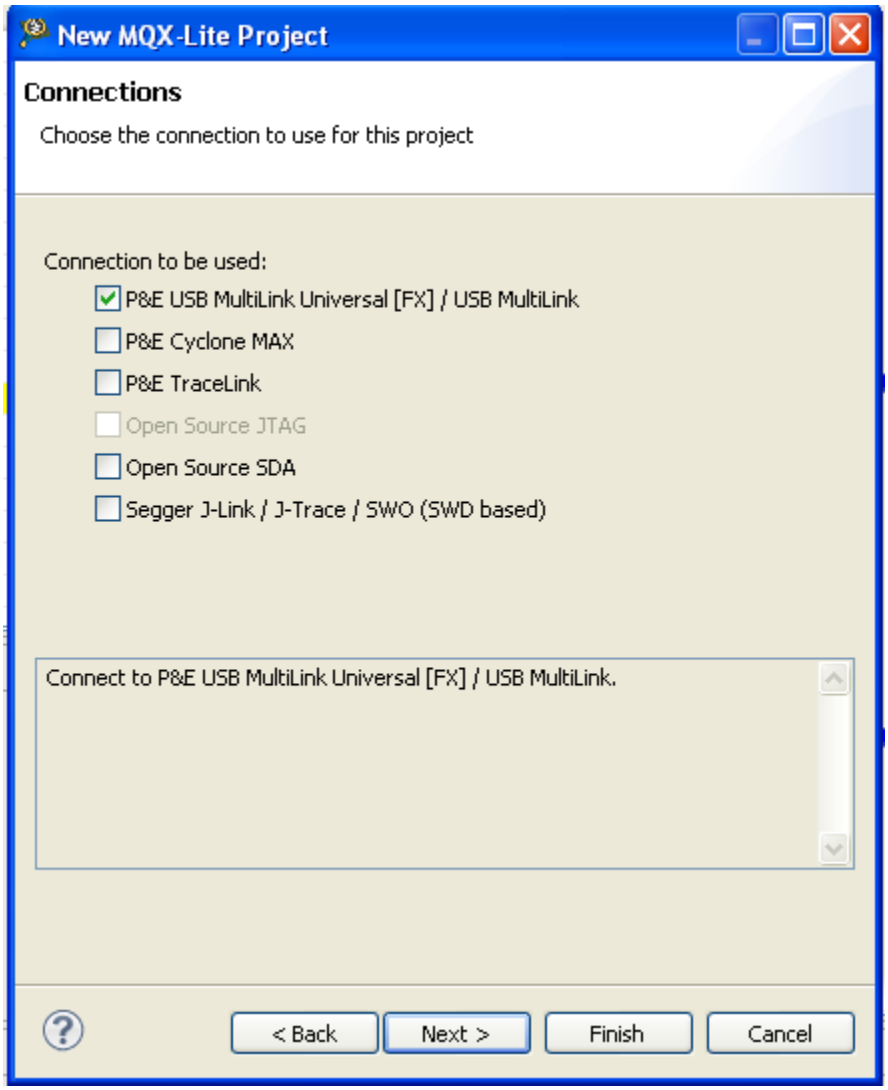

Figure 3-3. Connections Page

4. The **Language and Build Tools Options** page appears. Do not change the default settings.

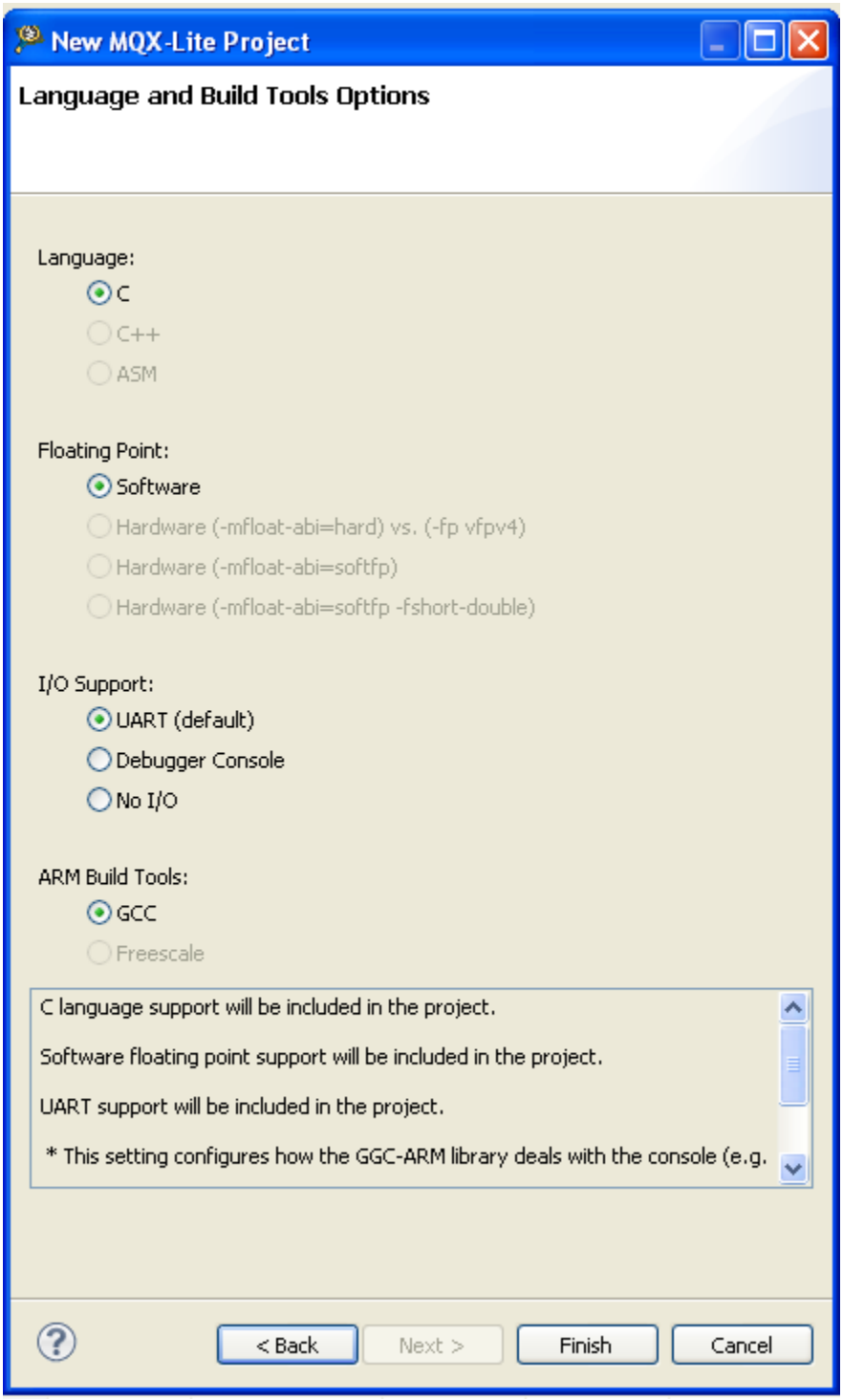

Figure 3-4. Language and Build Tools Options Page

5. Click **Finish** to create a project.

3.2 Adding MQX Lite into an Existing Project

The Processor Expert technology also enables adding MQX Lite RTOS to existing applications. The following steps describe how to add MQX Lite RTOS to an existing application.

1. Double-click on MQX Lite component in the component selector window to add the component to a project.

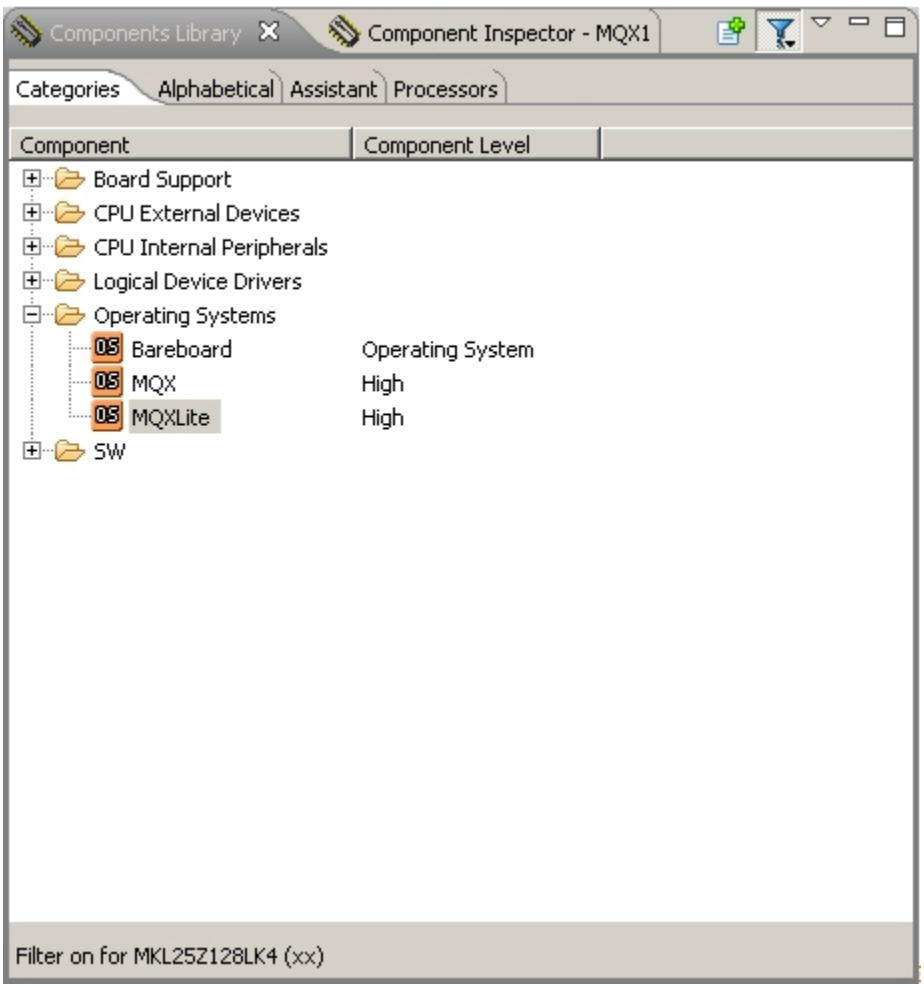

Figure 3-5. Components Library View

2. Configure MQX Lite component.

   Configure the MQX Lite operating system and define number of application tasks according to the application needs. For more information, refer to the MQX Lite Component Interface and MQXLite_task Component Interface sections.

3.3 Define Task Routines in your Application

Once MQX Lite components are configured properly, the code has to be generated. This is initiated by the *Generate Processor Expert Code* command accessible from the pop-up menu. For more information, see the *Processor Expert User Guide*.

If no issue occurs in code generation, all generated source code files are placed in the destination directory. MQX Lite task routine prototypes are generated into the `mqx_tasks.h` header file. Skeletons of all task routines, defined by the `MQXLite`/`MQXLite_task` components, are generated into the `mqx_tasks.c` source file (unless a different name was specified). The next step is to implement the functionality of each defined task routine before building the project.

3.4 MQX Lite Demo Applications

The easiest way how to start with MQX Lite operating system is through example applications.

CodeWarrior examples are available at: `CodeWarrior}\MCU\CodeWarrior_Examples \Processor_Expert\MQXLite` directory.

PEx Driver Suite examples are available at: `PExDriverSuite}\eclipse\ProcessorExpert \Projects\MQXLite` directory.

This table summarizes MQX Lite example applications available within the Processor Expert installation.

Table 3-1. MQX Lite Example Applications

Name	*Description*
I2C_RGB	MQXLite-based application that changes the RGB dimming, blink rate and the color as per the board tilt and the slider position.
LED	Shows how one task creates another one and how the task creation parameter is passed. Two LEDs are used to reflect the activity of created MQX Lite tasks (different frequencies of LED blinking).
LWDEMO	Shows MQX Lite multitasking and inter-process communication using the following lightweight components: LW Semaphores, LW Events and LW Message Queue.
LWEVENT	Simple demonstration of MQX Lite Lightweight Event component.
LWSEM	Simple demonstration of MQX Lite task synchronization using the Lightweight Semaphore component.
MUTEX	Simple demonstration of MQX Lite task synchronization using the Mutex component.

Note that not all MQX Lite example applications are available for all FSL development platforms.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXLITEUG
Rev. 1.1
02/2014