Document Number: MQXGSMULTICORE
Rev. 10, 02/2014

# Using Freescale MQX™ RTOS on Multicore Devices

| | |
|---|---|
| **PRODUCT:** | Freescale MQX™ RTOS |
| **PRODUCT VERSION:** | 4.1.0 |
| **DESCRIPTION:** | Using Freescale MQX™ RTOS on multicore devices, version 4.1.0 |
| **RELEASE DATE:** | February, 2014 |

***How to Reach Us:***

**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support



Document Number: MQXGSMULTICORE
Rev. 10, 02/2014

# Table of Contents

# 1 Support for multicore operation in MQX

## 1.1 Introduction

The MQX™ 3.8 introduces support for multicore operation on PowerPC platforms of MPX family having multiple cores of the same type.

## 1.2 Execution modes of multicore MPX parts

The multicore parts of MPX family may operate either in lock-step mode (LSM) or decoupled parallel mode (DPM). In LSM, both cores execute the same code at the same time (redundancy for safety applications) while in DPM the cores execute code independently on each other, so there are effectively two processors available. The mode is selected by LSM/DPM bit in shadow FLASH area. The mode cannot be changed in runtime. Therefore, a proper mode needs to be set prior to running an application on the MCU. For more information, see the part-specific notes in *Getting Started with Freescale MQX™ RTOS*.

In regards to the application, a part configured in LSM behaves as was a single processor. Therefore, other than the early initialization which is handled by the startup code in the BSP, it is virtually no different than running MQX on a single core part. This document describes an instance where the DPM requires the application to be designed in such a way as to take advantage of multicore execution. The subsequent content presumes that a dual-core system is used (e.g. PXS30).

# 2 Principle of operation

Current multicore operation of MQX involves starting multiple instances of MQX, one on each core. Each instance of MQX may use of different set of drivers typically servicing distinct set of peripherals and different set of tasks.

## 2.1 Memory map

Both instances of MQX are started from a single image. Therefore, while they share the same code memory, the data memory is separate for each instance. The principle of linking process requires the data memory to be mapped to the same address space on both cores. That means that the global variables in both instances of MQX are located at the same (virtual) address, but, in fact, they occupy different locations of physical memory. This is achieved by proper configuration of MMUs ,memory management units, which takes place in the startup code and BSP initialization.

Figure 1 illustrates the memory mapping. Please note that the illustration depicts the basic concept and does not cover details like shared memory for inter-processor communication (IPC), uncached memory for peripheral buffers, etc.

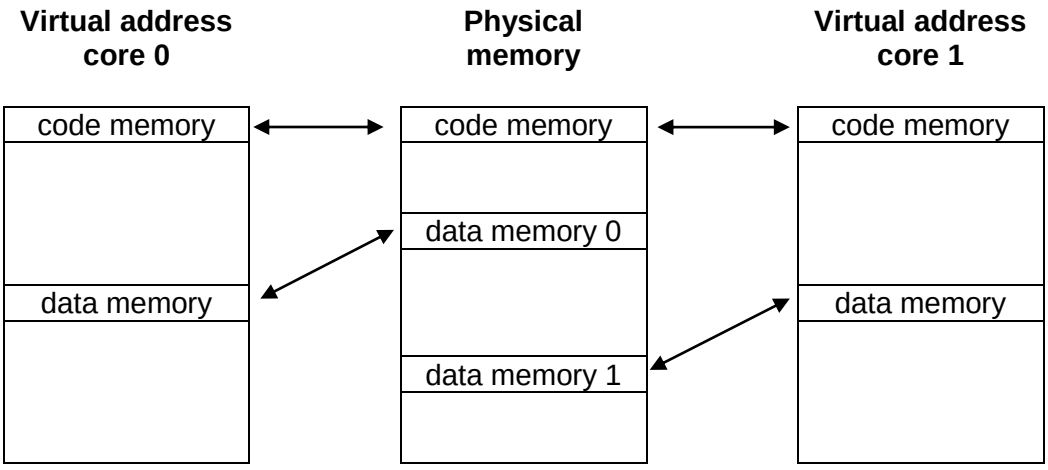


*Figure 1: multicore memory mapping*

# 3 Design and implementation of a multicore application

## 3.1 Entry point (main function) of a multicore BSP

The following code snippet shows the main function which is part of the BSP supporting multicore operation:

```
int main(void)
{
   extern const MQX_INITIALIZATION_STRUCT MQX_init_struct;
   extern const MQX_INITIALIZATION_STRUCT MQX_init_struct_1;

   /* Start MQX */
   if (_psp_core_num()==0) {
       _mqx( (MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct );
   } else {
       _mqx( (MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct_1 );
   }
   return 0;
}
```

Compared to a single-core operation, the main function contains conditional statement which is testing the core number the code is running on, and multiple calls to _mqx function with pointers to different initialization structures. Only one of the calls to _mqx function takes place on each core.

## 3.2 Selecting tasks to start on each core

On a single core device one or more tasks are defined upon MQX start according to MQX_template_list array:

```
const TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
/*
  Task Index, Function, Stack, Priority, Name, Attributes, Param, Time Slice
*/
    { 1, test_task,   1500, 8, "test_task",   MQX_AUTO_START_TASK, 0, 0 },
};
```

This basic concept is followed also on a multicore system. To start a task on the second core, there is a separate TASK_TEMPLATE_STRUCT with identifier MQX_template_list_1 defining tasks to run in the same fashion as for the first core.

```
const TASK_TEMPLATE_STRUCT MQX_template_list_1[] =
{
/*
  Task Index, Function, Stack, Priority, Name, Attributes, Param, Time Slice
*/
    { 1, test_task,   1500, 8, "test_task",   MQX_AUTO_START_TASK, 0, 0 },
    { 1, test_task_1, 1500, 8, "test_task_1", MQX_AUTO_START_TASK, 0, 0 },
};
```

If there is no MQX_template_list_1 defined, then no tasks will be started on the second core (besides the idle task). The second instance of MQX itself will be started in any case.

The sets of tasks executed on the cores need not be distinct. Multiple instances of the same task may be running on multiple cores if the task is designed for such an operation (e.g. handles concurrent access to HW devices, etc.). Please note that tasks running on different cores do not share the same global data (see Chapter 2). Inter-process communication (IPC) may be used to pass the data between tasks running on different cores.

## 3.3 Assigning HW devices to particular core

Typically, the recommendation is to have the instances of MQX running on different cores using distinct set of peripherals. Initialization of peripherals in the BSP is performed conditionally according to defined macros which contain masks of cores the particular peripheral should be used on. To test whether the peripheral is enabled or disabled on particular core, call PSP function _psp_core_peripheral_enable, which returns true only if executed on the core for which the corresponding bit of the parameter is set to 1.

Example:

```
if (_psp_core_peripheral_enabled(CORECFG_SPI_0)) {
    _dspi_polled_install("spi0:", &_bsp_dspi0_init);
}
```

Whereas CORECFG_SPI_0 is defined:

```
#define CORECFG_SPI_0          CORE_0
```

Or:

```
#define CORECFG_SPI_0          CORE_1
```

Or, in special cases, if the same peripheral is to be used on both cores:

```
#define CORECFG_SPI_0          (CORE_0|CORE_1)
```

The values of CORECFG may be defined in user_config.h. Otherwise, default values from PSP are used.

The test using the _psp_core_peripheral_enabled function may be also done in BSP specific module of a driver. Typically, this is the case when it is necessary to conditionally enable access to a given peripheral in the peripheral bridge (e.g. _bsp_dspi_enable_access function).

## 3.4 Inter-core synchronization

Synchronization of tasks running on different cores is supported by the core_mutex driver providing mutual exclusion mechanism between tasks which are running on different cores.

### 3.4.1 SEMA4 peripheral

MPX family devices feature SEMA4 units containing gates with mutual exclusion mechanism and ability to notify core(s) by an interrupt when the gate is unlocked which provides the efficient way to unblock a waiting task without needing a busy loop checking for locked/unlocked status.

The SEMA4 unit is used as an underlying device by the core_mutex driver. There may be several SEMA4 units, one per core, each having multiple gates with mutual exclusion mechanism.

### 3.4.2 Description of core_mutex API functions

Full prototypes of the functions may be found in source/io/core_mutex/core_mutex.h

**_core_mutex_install(const CORE_MUTEX_INIT_STRUCT * init_ptr)**

The function performs initial installation of the device. This is done only once on each core, typically upon system initialization in the BSP.

**_core_mutex_create(uint32_t dev_num, uint32_t mutex_num, uint32_t policy)**

The function allocates core_mutex structure and returns a handle to it. The mutex to be created is identified by SEMA4 and mutex (gate) number. The handle is used to reference the created mutex in calls to other core_mutex API functions. This function is to be called only once for each mutex. The policy parameter determines behavior of task queue associated with the mutex.

**_core_mutex_create_at(CORE_MUTEX_PTR mutex_ptr, uint32_t dev_num, uint32_t mutex_num, uint32_t policy)**

Function similar to _core_mutex_create but it does not use dynamic allocation of CORE_MUTEX structure. A pointer to pre-allocated memory area is passed by the caller instead.

**_core_mutex_get(uint32_t dev_num, uint32_t mutex_num)**

Returns handle to an already created mutex.

**_core_mutex_lock(CORE_MUTEX_PTR core_mutex_ptr)**

The function attempts to lock a mutex. If the mutex is already locked by another task the function blocks and waits until it is possible to lock the mutex for the calling task.

**_core_mutex_trylock(CORE_MUTEX_PTR core_mutex_ptr)**

The function attempts to lock a mutex. If the mutex is successfully locked for the calling task, COREMUTEX_LOCKED is returned. If the mutex is already locked by another task, the function does not block but rather returns COREMUTEX_UNLOCKED immediately.

**_core_mutex_unlock(CORE_MUTEX_PTR core_mutex_ptr)**

Unlocks the given mutex.

**_core_mutex_owner(CORE_MUTEX_PTR core_mutex_ptr)**

Returns the number of core currently “owning” the mutex.

### 3.4.3 Example of core_mutex usage

The following code snippet shows usage of core_mutex API. The code presumes that _core_mutex_install is already called which typically takes place during BSP initialization.

```
void test_task(uint32_t initial_data)
{
    CORE_MUTEX_PTR cm_ptr;

    cm_ptr = _core_mutex_create( 0, 1, MQX_TASK_QUEUE_FIFO );

    while (1) {
        _core_mutex_lock(cm_ptr);
        /* mutex locked here */
        printf("Core%d mutex locked\n", _psp_core_num());
        _time_delay((uint32_t)rand() % 20 );
        _core_mutex_unlock(cm_ptr);
        /* mutex unlocked here */
        printf("Core%d mutex unlocked\n", _psp_core_num());
        _time_delay((uint32_t)rand() % 20 );
    }
}
```

## 3.5 Inter-processor communication

The inter-processor communication (IPC) may be used to pass data between tasks running on different cores. For actual data transport a PCB (packet control block) device is used, particularly PCB using shared memory area. For generic information concerning IPC operation, see *MQX User's Guide*.

### 3.5.1 Example of IPC setup on a dual-core device

There are two instances of a routing table, one per core:

```
const IPC_ROUTING_STRUCT core0_routing_table[] =
{
   { BSP_CORE_1_PROCESSOR_NUMBER, BSP_CORE_1_PROCESSOR_NUMBER, QUEUE_TO_REMOTE
},
   { 0, 0, 0 }
};

const IPC_ROUTING_STRUCT core1_routing_table[] =
{
   { BSP_CORE_0_PROCESSOR_NUMBER, BSP_CORE_0_PROCESSOR_NUMBER, QUEUE_TO_REMOTE
},
   { 0, 0, 0 }
};
```

PCB over shared memory will be used as transport layer for IPC. Therefore, init structures are needed. A structure which describes shared memory comes first.

```
const IO_PCB_SHM_INIT_STRUCT pcb_shm_init =
{
    /* TX_BD_ADDR       */ BSP_SHARED_RAM_START,
    /* TX_LIMIT_ADDR    */ (unsigned char*)(BSP_SHARED_RAM_START)+1024,
    /* RX_BD_ADDR       */ BSP_REMOTE_SHARED_RAM_START,
    /* RX_LIMIT_ADDR    */ (unsigned char*)(BSP_REMOTE_SHARED_RAM_START)+1024,
    /* INPUT_MAX_LENGTH */ 128,
    /* RX_VECTOR        */ MPXS30_INTC_SSCIR0_VECTOR,
    /* TX_VECTOR        */ MPXS30_INTC_SSCIR1_VECTOR,
    /* REMOTE_RX_VECTOR */ MPXS30_INTC_SSCIR0_VECTOR,
    /* REMOTE_TX_VECTOR */ MPXS30_INTC_SSCIR1_VECTOR,
    /* INT_TRIGGER      */ _bsp_io_pcb_shm_int_trigger
};
```

Generic PCB init structure references a shared memory init structure.

```
const IPC_PCB_INIT_STRUCT pcb_init =
{
   /* IO_PORT_NAME */              "pcb_shmem:",
   /* DEVICE_INSTALL? */           _io_pcb_shm_install,
   /* DEVICE_INSTALL_PARAMETER*/   (void*)&pcb_shm_init,
   /* IN_MESSAGES_MAX_SIZE */      sizeof(THE_MESSAGE),
   /* IN_MESSAGES_TO_ALLOCATE */   8,
   /* IN_MESSAGES_TO_GROW */       8,
   /* IN_MESSAGES_MAX_ALLOCATE */ 16,
   /* OUT_PCBS_INITIAL */          8,
   /* OUT_PCBS_TO_GROW */          8,
   /* OUT_PCBS_MAX */              16
};
```

Protocol init structure defines that PCB will be used as a transport layer for IPC.

```
const IPC_PROTOCOL_INIT_STRUCT ipc_init_table[] =
{
   { _ipc_pcb_init, (void*)&pcb_init, "core_ipc_pcb", QUEUE_TO_REMOTE },
   { NULL, NULL, NULL, 0}
};
```

There are two instances of IPC init structure, one per core.

```
const IPC_INIT_STRUCT core0_ipc_init = {
    core0_routing_table,
    ipc_init_table
};

const IPC_INIT_STRUCT core1_ipc_init = {
    core1_routing_table,
    ipc_init_table
};
```

The _ipc_task has to be an autostart task on both cores. Pointer to IPC initialization structure for given core is passed as CREATION_PARAMETER to _ipc_task.

```
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
    { IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 8, "_ipc_task",
MQX_AUTO_START_TASK, (uint32_t) &core0_ipc_init,     0 },
    /* application specific tasks here */
    { 0 }
};

TASK_TEMPLATE_STRUCT  MQX_template_list_1[] =
{
    { IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 8, "_ipc_task",
MQX_AUTO_START_TASK, (uint32_t) &core1_ipc_init,     0 },
    /* application specific tasks here */
    { 0 }
};
```

Now, tasks running on different cores may communicate with each other by passing messages.

```
my_qid   = _msgq_open(MAIN_QUEUE,0);
msgpool = _msgpool_create(sizeof(THE_MESSAGE), 8, 8, 0);
msg_ptr = (THE_MESSAGE_PTR)_msg_alloc(msgpool);

if (msg_ptr != NULL) {
    msg_ptr->HEADER.TARGET_QID =
        _msgq_get_id((_processor_number)dest_core,REMOTE_QUEUE);
    msg_ptr->HEADER.SOURCE_QID = my_qid;
    msg_ptr->DATA = 0;
   _msgq_send(msg_ptr);
}

…

msg_ptr = _msgq_receive(MSGQ_ANY_QUEUE,0);
```

See “Messages” section of the MQX User Guide for details concerning message queues.