Using Freescale MQX™ RTOS on multi-core devices

PRODUCT:	Freescale MQX™ RTOS
PRODUCT VERSION:	3.8.0
DESCRIPTION:	Using Freescale MQX™ RTOS on multi-core devices, version 3.8.0
RELEASE DATE:	February 15th, 2012

freescale™ semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan**:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Rev. 08
12/2011

Table of Contents

1 Support for multi-core operation in MQX

1.1 Introduction

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

1.2 Execution modes of multi-core MPX parts

The multi-core 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, thus proper mode needs to be set prior running application on the MCU, please refer to part specific notes in the getting started document.

From the application point of view, part configured in LSM behaves like it was just a single processor so there is virtually no difference to running MQX on a single core part (besides early initialization handled by startup code in the BSP). This document rather deals with the other case, the DPM which requires the application to be designed accordingly to take advantage of multi-core execution. The following text presumes a dual-core system is used (e.g. PXS30).

2 Principle of operation

Current multi-core 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 services distinct set of peripherals and different set of tasks.

2.1 Memory map

Both instances of MQX are started from a single image, thus they share the same code memory, while 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 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.

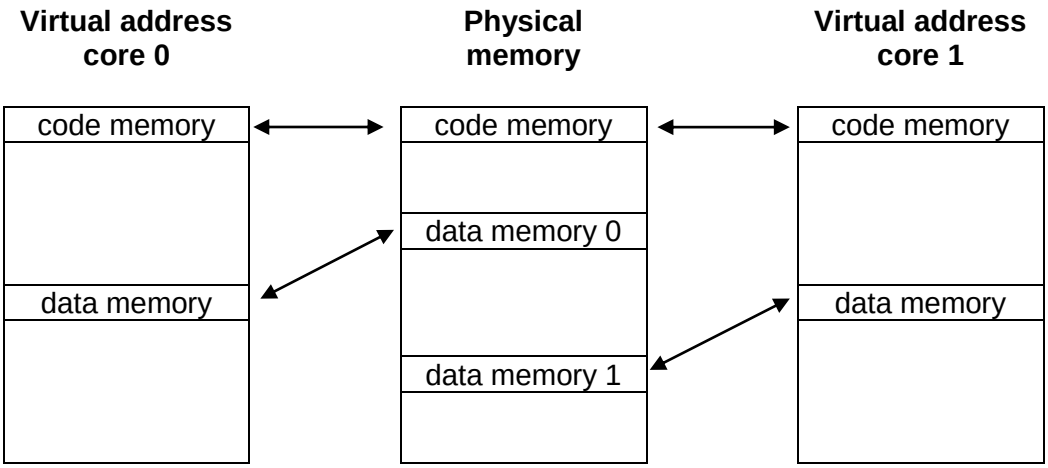

Figure 1: multi-core memory mapping

3 Design and implementation of a multi-core application

3.1 Entry point (main function) of a multi-core BSP

The following code snippet shows the main function which is part of BSP supporting multi-core 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 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 the same way also in the case of a multi-core system. For starting task on 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). However, the second instance of MQX itself will be started in any case.

The sets of tasks executed on the cores need not to be distinct. Multiple instances of the same task may be running on multiple cores, if the task is designed for such 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 data between tasks running on different cores.

3.3 Assigning HW devices to particular core

Typically it is desired that the instances of MQX running on different cores use distinct set of peripherals. Initialization of peripherals in BSP is performed conditionally according to defined macros containing masks of cores the particular peripheral should be used on. The test whether the peripheral is enabled or disabled on particular core is done by calling PSP function _psp_core_peripheral_enable, which returns true only if executed on core for which 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 needed to conditionally enable access to 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 taks 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 a gate is unlocked which provides with efficient way to unblock a waiting task without necessity of 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 BSP.

_core_mutex_create(uint_32 core_num, uint_32 mutex_num, uint_32 policy)

The function allocates core_mutex structure and returns a handle to it. The mutex to be created is identified by core 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, ….)

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(uint_32 core_num, uint_32 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, TRUE is returned. If the mutex is already locked by another task the function does not block but rather returns FALSE immediately.

_core_mutex_unlock(CORE_MUTEX_PTR core_mutex_ptr)

Unlocks the given mutex.

_core_mutex_owner(CORE_MUTEX_PTR core_mutex_ptr)

Returns number of core number 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(uint_32 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((uint_32)rand() % 20 );
        _core_mutex_unlock(cm_ptr);
        /* mutex unlocked here */
        printf("Core%d mutex unlocked\n", _psp_core_num());
        _time_delay((uint_32)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, please refer to MQX User's guide.

3.5.1 Example of IPC setup on a dual-core device

There are two instances of 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, thus init structures are needed. Structure describing shared memory comes first.

```
const IO_PCB_SHM_INIT_STRUCT pcb_shm_init =
{
    /* TX_BD_ADDR        */ BSP_SHARED_RAM_START,
    /* TX_LIMIT_ADDR     */ (uchar_ptr)(BSP_SHARED_RAM_START)+1024,
    /* RX_BD_ADDR        */ BSP_REMOTE_SHARED_RAM_START,
    /* RX_LIMIT_ADDR     */ (uchar_ptr)(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 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*/   (pointer)&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, (pointer)&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, (uint_32) &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, (uint_32) &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 MQX User’s manual for details concerning message queues.