electronic design
The Authority on Emerging Technologies for Design Solutions

Porting OpenMCAPI For Multicore Communications

Ruslan Filipovich
Tue, 2013-06-18 14:07

In their routine work, embedded system developers often have to launch two or more diverse operating systems (OSs) on n-core systems-on-chip (SoCs). The OSs usually include Linux and a specialized real-time operating system (RTOS). In this case, Linux has to handle the burden of heavy protocol stacks, while an RTOS manages real-time tasks.

One of the main tasks related to such a system structure is to provide a mechanism for communication, e.g., inter-core data exchange. Developers can solve the problem by using shared memory and inter-core interruptions by writing their own interaction interlayer and porting it to different OSs.

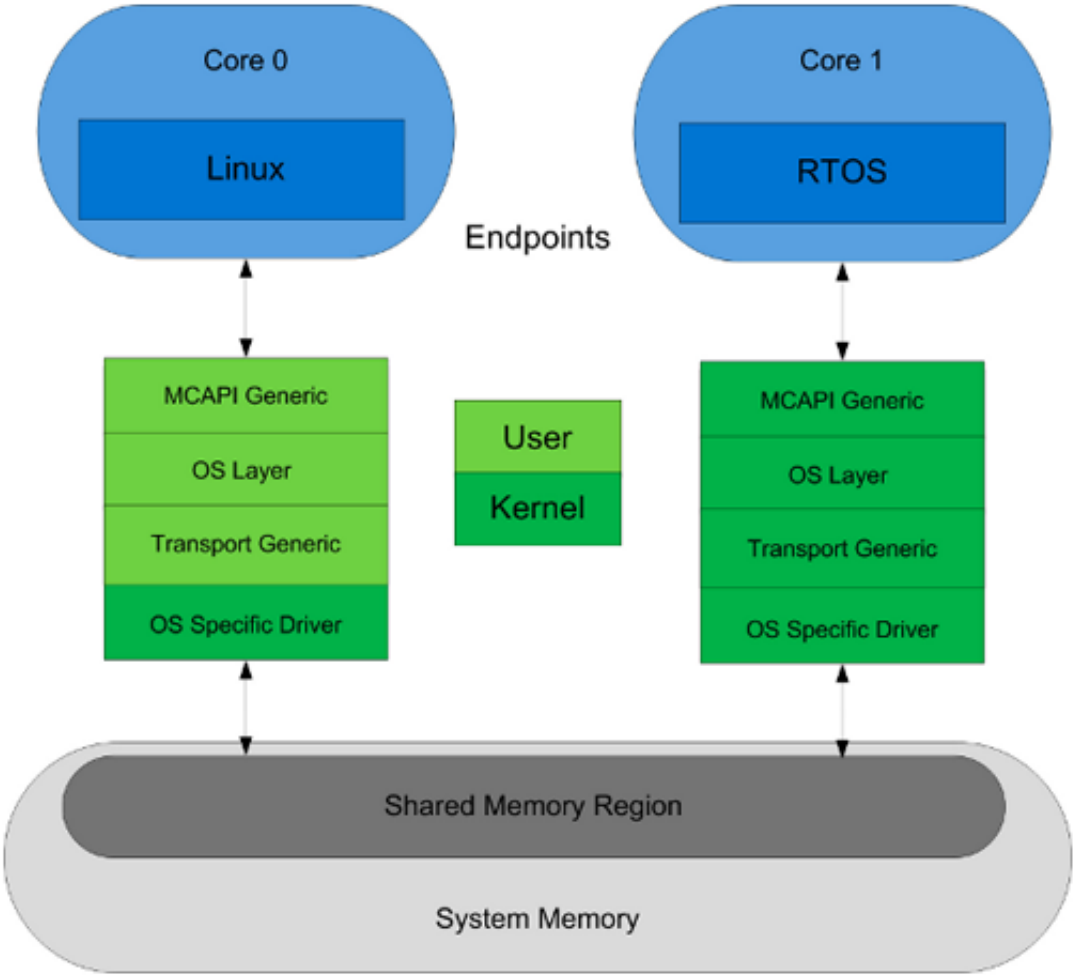
About the Author
Ruslan Filipovich, programmer
Promwad Innovation Company
http://www.promwad.com

Ruslan develops software for electronic devices based on Embedded Linux, Android and RTOS. He implemented projects for portable navigation devices (GPS/GLONASS), power management systems and plug-computers. Now he develops embedded software for multi-core platforms.

To bring this application programming interface (API) to the standardized format, the Multicore Association (MCA) developed and released the first version of the specification Multicore Communications API (MCAPI). The second version soon followed.

OpenMCAPI, the library in question, is based on the MCAPI 2.0 specification developed by Mentor Graphics. It has an open-source code under a free Berkeley Software Distribution (BSD) or GNU Public License (GPL) license. The OpenMCAPI source code provides brief information on launching and porting. The OpenMCAPI library initially enables developers to work under Linux using a virtual transport or shared memory, but only on the mpc85xx and mv78xx0 platforms.

Figure 1 shows the proposed structure of communication between Linux and the RTOS through OpenMCAPI with division into abstract levels. MCAPI Generic is an implementation of the external MCAPI API. OS Layer is part of the MCAPI Generic level, which contains a code specific to the OS. Described in the file libmcapi/mcapi/linux/mcapi_os.c, this part implements the work of a system thread, mutexes, queue management functions, and lockable delays.



Transport Generic is an abstraction level that provides a mechanism for shared memory at the user space level. Represented by the files libmcapi/shm/shm.c and libmcapi/shm/linux/shm_os.c, it implements shared memory mapping. It also sends notifications and receives messages using core system calls. The file libmcapi/shm/sysv.c is used on the platform without the support of transport shared memory to virtualize the transport required for OpenMCAPI demonstration and testing.

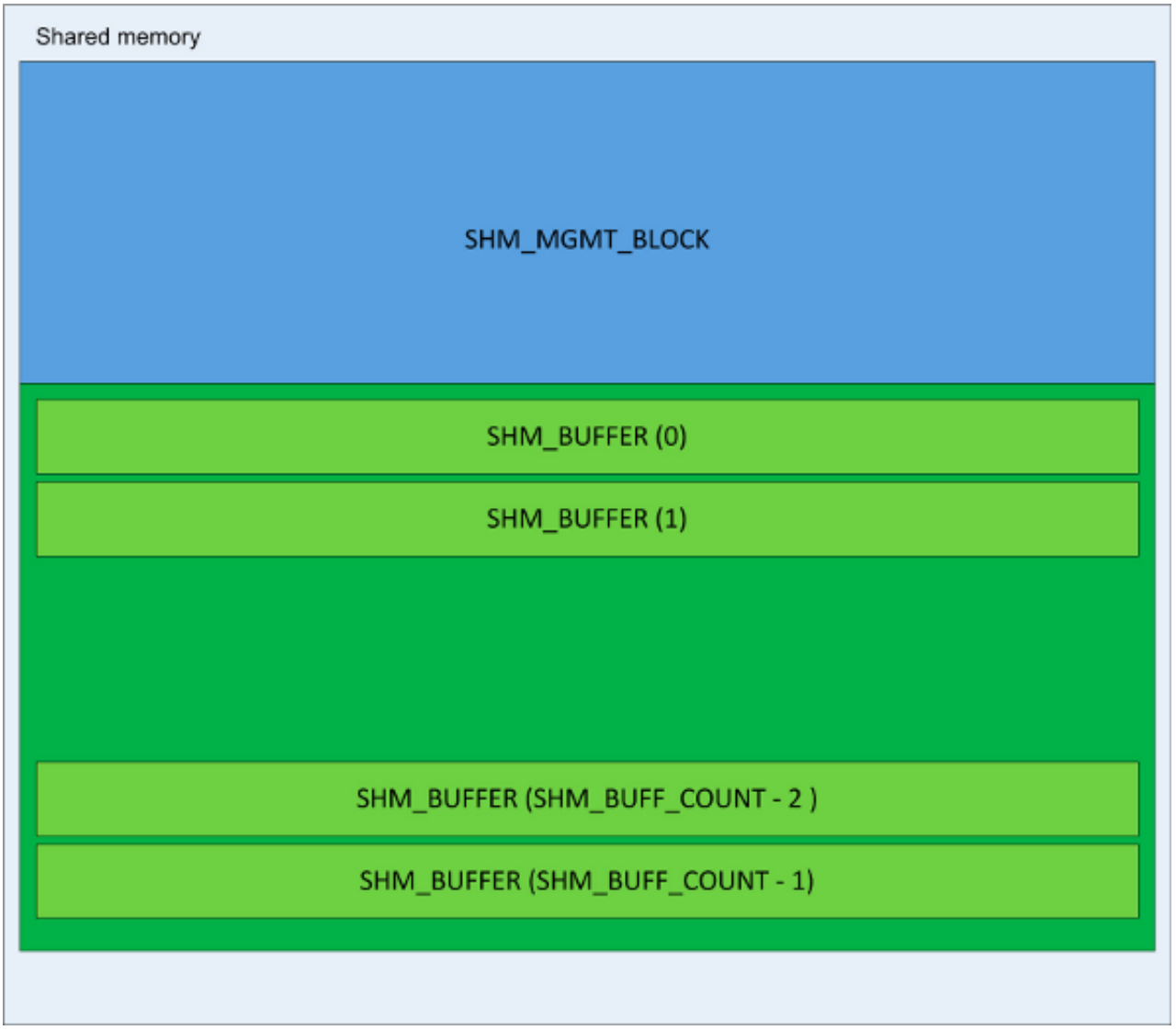
Enabling transport virtualization eliminates the need for an OS-specific driver. Instead, a Linux kernel module that provides direct access to the hardware from the user space represents the OS-specific driver. Located in the folder libmcapi/shm/linux/kmod, the module includes the implementation of API interrupt handlers and user space (shared memory mapping, sending interrupts), which are implemented in the mcomm.ko kernel module, and hardware-specific code for a specific platform (mpc85xx or mv78xx0).

To fully understand the mechanism of communication through transport using shared memory, which is implemented in the OpenMCAPI library, we should examine the inter-core signaling mechanism and the structure of data in shared memory.

Further discussion will involve the mpc85xx platform (the P1020 chip by Freescale) and Linux kernel v. 2.6.35 with patches, which comes with the SDK Freescale QorIQ_SDK_V1_03 software development kit (SDK). Real-time operating system Real-Time Executive for Multiprocessor Systems (RTEMS) source code can be obtained in the RTEMS git repository.

To implement inter-core signaling, Freescale provides at least two mechanisms: interprocessor interrupts (IPIs), which are inter-core interrupts, with up to four multicast-enabled interrupts, and message interrupts (MSGRs), which are inter-core 32-bit messages generating up to eight interrupts when the message is recorded in the register.

With an implementation of the OS-specific driver, the OpenMCAPI library uses the MSGRs mechanism for this platform. Now, let's consider the structure of the data contained in shared memory *(Fig. 2)*.



We can divide the shared memory region into two blocks, depending on the use of space. The first block is the SHM_MGMT_BLOCK region represented by the following structure:

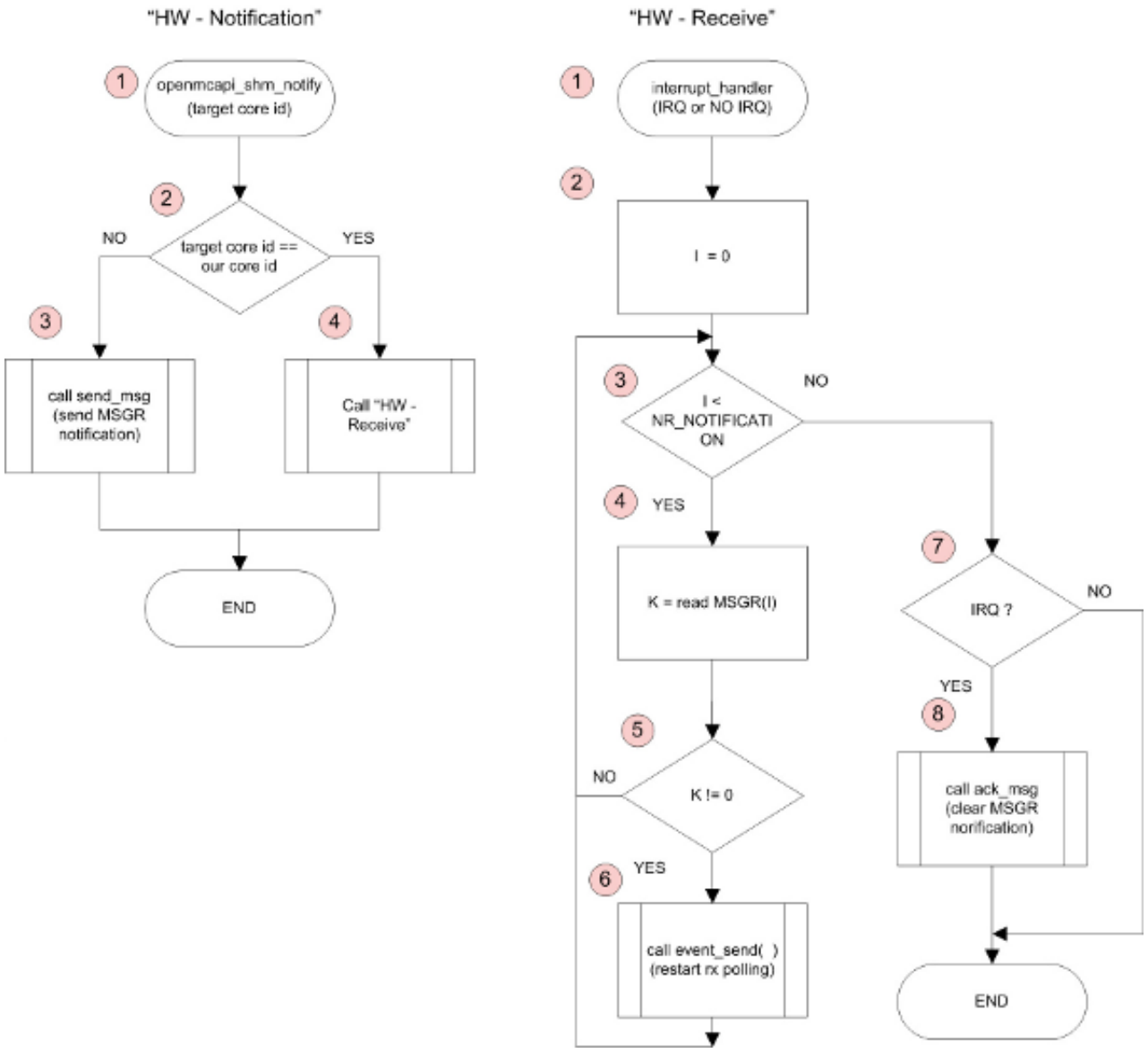
```
1.  /* SM driver mamagement block */
2.  struct _shm_drv_mgmt_struct_
3.  {
4.      shm_lock                    shm_init_lock;
5.      mcapi_uint32_t              shm_init_field;
6.      struct _shm_route_          shm_routes[CONFIG_SHM_NR_NODES];
7.      struct _shm_buff_desc_q_    shm_queues[CONFIG_SHM_NR_NODES];
8.      struct _shm_buff_mgmt_blk_  shm_buff_mgmt_blk;
9.  };
```

The structure includes five elements:

- shm_init_lock is global locking of shared memory, used to delimit n-core access to the shared area.
- The variable shm_init_field contains the key to the end of initialization of the master2. At the end of initialization it takes the value SHM_INIT_COMPLETE_KEY.
- Shm_routes is a routing table with inter-core message connections. It includes CONFIG_SHM_NR_NODES connections based on the number of those involved in core (node) exchange. In our case there are two nodes.

- Shm_queues are message queues with reference to a specific node. They include CONFIG_SHM_NR_NODES. In our case there are two queues.
- Shm_buff_mgmt_blk is a buffer management structure (SHM_BUFFER) in the data region.

The second block is the data region, which includes SHM_BUFF_COUNT (128 by default) SHM_BUFFER structures. This area is used directly for the storage of transmitted data. The SHM_BUFFER structure consists of an MCAPI_MAX_DATA_LEN size array and an additional element management structure. Before we examine the porting process, though, we should introduce a block chart of the low-level mechanism of communication through shared memory *(Fig. 3)*.



HW-Notification describes the process of sending a notification to the remote or current core. The function call accepts the identification of the core for which the message is intended (the function OpenMCAPI_shm_notify). If the notification is intended for the remote core, "target id," a deleted message is generated through the MSGRs mechanism with the data field value equal to one (Block 3) or is an explicit call to the interrupt handler interrupt_handler ("HW-Receive," Block 4).

HW-Receive describes the process of receiving a notification from the remote or current core. Interrupt_handle is an interrupt handler set to operate when receiving a message through MSGRs. It is also used for an explicit call. Blocks 2-6 show how the status of all MSGRs messages is checked. If the MSGR message data field is not equal to 0, it leads to unlocking the thread, which processes the data in the shared memory area. Blocks 7-8 show how the call site of the interrupt handler is checked ("interrupt_handler"). If the call was interrupted, the MSGR message flag is reset.

RTEMS is a full-featured RTOS with an open source code. It supports multiple open standard APIs, Portable Operating System Interface for Unix (POSIX) standards, and BSD sockets. Designed for use in space, medical, networking, and many other embedded devices, RTEMS provides a wide range of processor architectures such as ARM, PowerPC, Intel, Blackfin, MIPS, and Microblaze. It contains a large stack of implemented network protocols, particularly tcp/ip, http, ftp, and telnet. RTEMS provides uniform access to RTC, NAND, UART, and other equipment.

The OpenMCAPI porting process implements the OS layer (libmcapi/mcapi/rtems/mcapi_os.c and libmcapi /include/rtems/mgc_mcapi_impl_os.h) and support for compatible shared memory transport (libmcapi/shm/rtems/shm_os.c). It also adds recipes to the waf build tool used for OpenMCAPI.

Since the P1020 (powerpc, 500v2) is the target platform and the porting was performed to the RTOS allowing for the absence of division of the kernel/user space, there is no need to write libmcapi/include/arch/powerpc/atomic.h or libmcapi/shm/rtems/kmod/. Moreover, there is now no need to implement the OS layer because RTEMS supports POSIX-compatible calls. The files mcapi_os.c and mgc_mcapi_impl_os.h were simply copied from the implementation for Linux.

The file shm_os.c includes an implementation of the shared memory transport, which includes call adaptation and an implementation of an exchange mechanism through MSGRs. Six functions need implementation:

mcapi_status_t OpenMCAPI_shm_notify (mcapi_uint32_t unit_id, mcapi_uint32_t node_id): This function sends a notification to the remote core or cores. The source code is:

```
1.  /* send notify remote core */
2.  mcapi_status_t OpenMCAPI_shm_notify(mcapi_uint32_t unit_id,
3.                                      mcapi_uint32_t node_id)
4.  {
5.          mcapi_status_t mcapi_status = MCAPI_SUCCESS;
6.          int rc;
7.
8.          rc = shm_rtems_notify(unit_id);
9.  if (rc) {
10. mcapi_status = MGC_MCAPI_ERR_NOT_CONNECTED;
11.         }
12.         return mcapi_status;
13. }
14.
15. static inline int shm_rtems_notify(const mcomm_core_t target_core)
16. {
17.         struct mcomm_qoriq_data *const data = &mcomm_qoriq_data;
18.         /* If the target is the local core, call the interrupt handler directly.
    */
19.         if (target_core == mcomm_qoriq_cpuid()) {
20.                 _mcomm_interrupt_handler(NO_IRQ, data);
21.         } else {
22.                 mcomm_qoriq_notify(target_core);
23.         }
24.         return 0;
25. }
26.
27. /* Wake up the process(es) corresponding to the mailbox(es) which just received
28.  * packets. */
29. static int _mcomm_interrupt_handler(rtems_vector_number irq, struct
    mcomm_qoriq_data *data)
30. {
31.         register int i;
32.         void *mbox = data->mbox_mapped;
33.         for (i = 0; i < data->nr_mboxes; i++) {
34.                 int active;
```

```
35.                 switch (data->mbox_size) {
36.                 case 1:
37.                         active = readb(mbox);
38.                         break;
39.                 case 4:
40.                         active = readl(mbox);
41.                         break;
42.                 default:
43.                         active = 0;
44.                 }
45.                 if (active) {
46.                         LOG_DEBUG("%s: waking mbox %d\n", __func__, i);
47. (void) rtems_event_send( data->rid, MMCAPI_RX_PENDING_EVENT );
48.                 }
49.                 mbox += data->mbox_stride;
50.         }
51.         if (irq != NO_IRQ) {
52.                 mcomm_qoriq_ack();
53.         }
54.         return 0;
55. }
```

mcapi_status_t OpenMCAPI_shm_os_init(void): This function creates and starts a low-level data reception thread. It's implemented by calling the functions rtems_task_create and rtems_task_start. The source code is:

```
1.  /* Now that SM_Mgmt_Blk has been  initialized, we can start the RX thread. */
2.  mcapi_status_t OpenMCAPI_shm_os_init(void)
3.  {
4.          struct mcomm_qoriq_data *const data = &mcomm_qoriq_data;
5.          rtems_id id;
6.          rtems_status_code sc;
7.
8.          if( RTEMS_SELF != data->rid ) {
9.                  return MCAPI_ERR_GENERAL;
10.         }
11.
12.         sc = rtems_task_create(
13.           rtems_build_name( 'S', 'M', 'C', 'A' ),
14.           MMCAPI_RX_TASK_PRIORITY,
15.           RTEMS_MINIMUM_STACK_SIZE,
16.           RTEMS_DEFAULT_MODES,
17.           RTEMS_DEFAULT_ATTRIBUTES,
18.           &id);
19.         if( RTEMS_SUCCESSFUL != sc ) {
20.                 return MCAPI_ERR_GENERAL;
21.         }
22.
23.         /* global save task id */
24.         data->rid = id;
25.
26.         sc = rtems_task_start( id, mcapi_receive_thread, 0 );
27.         if( RTEMS_SUCCESSFUL != sc ) {
28.                 perror( "rtems_task_start\n" );
29.                 return MCAPI_ERR_GENERAL;
30.         };
31.
32.         return MCAPI_SUCCESS;
33. }
34.
35. static rtems_task mcapi_receive_thread(rtems_task_argument argument)
36. {
37.  int rc;
38.  do {
39.         rc = shm_rtems_wait_notify(MCAPI_Node_ID);
40.                 if (rc < 0) {
41.                         perror("shm_rtems_wait_notify");
```

```
                        break;
                }

                MCAPI_Lock_RX_Queue();
         /* Process the incoming data. */
                shm_poll();
                MCAPI_Unlock_RX_Queue(0);
} while (1);
printk("%s exiting!\n", __func__);
}

static inline int shm_rtems_wait_notify(const mcapi_uint32_t unitId)
{
        rtems_event_set event_out;
        int ret = 0;

        while(1) {
                LOG_DEBUG("mcomm_mbox_pending start\n");

                (void) rtems_event_receive(
                                        MMCAPI_RX_PENDING_EVENT,
                                        RTEMS_DEFAULT_OPTIONS,
                                        RTEMS_NO_TIMEOUT,
                                        &event_out
                                    );
                LOG_DEBUG("rtems_event_receive\n");

                ret = mcomm_mbox_pending(&mcomm_qoriq_data,
                                         (mcomm_mbox_t)unitId);

                LOG_DEBUG("mcomm_mbox_pending end ret=%d\n", ret);
                if(ret != 0) {
                        return ret;
                };
        }
        return 0;
}
```

mcapi_status_t OpenMCAPI_shm_os_finalize(void): This function stops the low-level data reception thread. It is implemented by calling the function rtems_task_delete. The source code is:

```
/* Finalize the SM driver OS specific layer. */
mcapi_status_t OpenMCAPI_shm_os_finalize(void)
{
        struct mcomm_qoriq_data *const data = &mcomm_qoriq_data;
        rtems_id id = data->rid;
        rtems_status_code sc;

        sc = rtems_task_delete(id);
        if( RTEMS_SUCCESSFUL != sc ) {
                return MCAPI_ERR_GENERAL;
        }

        return MCAPI_SUCCESS;
}
```

• void *OpenMCAPI_shm_map(void): This function involves preparing and setting the MSGRs interface, as well as preparing shared memory. The source code is:

```
/* full open mcom device and get memory map address*/
void *OpenMCAPI_shm_map(void)
{
        void *shm;
        int rc;
        size_t shm_bytes;

```

```
        // low level init //
        mcomm_qiroq_probe();
        shm_bytes = shm_rtems_read_size();
        if (shm_bytes <= 0) {
                        perror("read shared memory size\n");
                        return NULL;
        }

        /* initialized device. */
        rc = shm_rtems_init_device();
 if (rc < 0) {
        perror("couldn't initialize device\n");
goto out;
 }

        shm = shm_rtems_read_addr();
        if (shm == NULL) {
                        perror("mmap shared memory");
                        goto out;
        }
        return shm;
out:
        return NULL;
}

static size_t shm_rtems_read_size(void)
{
        struct mcomm_qoriq_data *const data = &mcomm_qoriq_data;
        return (size_t) (data->mem.end - data->mem.start);
}

static inline int shm_rtems_init_device(void)
{
        struct _shm_drv_mgmt_struct_ *mgmt = NULL; /* xmmm */
        return mcomm_dev_initialize(&mcomm_qoriq_data,
    (uint32_t)&mgmt->shm_queues[0].count,
            CONFIG_SHM_NR_NODES,
   sizeof(mgmt->shm_queues[0].count),
            ((void *)&mgmt->shm_queues[1].count - (void *)&mgmt-
>shm_queues[0].count));
}

static void *shm_rtems_read_addr(void)
{
        struct mcomm_qoriq_data *const data = &mcomm_qoriq_data;
        return (void*)data->mem.start;
}
```

• void OpenMCAPI_shm_unmap(void *shm): This function closes the MSGRs interface and reverts the use of shared memory. The source code is:

```
/* full close mcom device and revert memory */

void OpenMCAPI_shm_unmap(void *shm)
{
  /* deinitialized device. */
  shm_rtems_deinit_device();

  // low level deinit //
  mcomm_qoriq_remove();
}

static inline int shm_rtems_deinit_device(void)
{
  return mcomm_dev_finalize(&mcomm_qoriq_data);
}
```

We should pay special attention to the implementation of the function of a low-level reception thread, mcapi_receive_thread. (See the source code above.) When you start the thread by calling the function rtems_event_receive, it switches to event standby mode (implemented by the event mechanism available in RTEMS). Then when the startup event arrives, it is sent to the interrupt_handler *(Fig. 3, HW-Receive chart)*. It processes changes in shared memory (calling the internal function OpenMCAPI - shm_poll ()) and places a provisional lock on it. After this, the thread returns to standby mode.

The table shows the results of communication between Linux and RTEMS through OpenMCAPI. The Freescale P1020RDB-PB development board with the installed processor P1020 (two cores) serves as a testing stand. The core frequency is 800 MHz, DDR2 is 400 MHz, and CCB is 400 MHz. We launched Linux/RTEMS on the cores 0/1 respectively. It was a two-way exchange and we measured the time required for 10,000 two-way sendings.

We can conclude that the OpenMCAPI library is an excellent implementation of the MCAPI specification. It has a clear structure of the source code, making porting easier. Moreover, it includes porting illustrations (PowerPC and Arm platforms), a free license, and a capacity sufficient for most applications.

Source URL: http://electronicdesign.com/dev-tools/porting-openmcapi-multicore-communications