Document Number: MQXMCCUG
Rev 2.1

MultiCore Communication Library
User Guide

Contents

1 Introduction

The MultiCore Communication, MCC, is a subsystem which enables applications to run on different cores in the multicore system. For example, MCC enables the communication between applications running on ARM® Cortex®-A5 and ARM® Cortex®-M4 cores on the Vybrid platform.

Features

- Lightweight, fast
- Uses shared RAM and interrupts
- Configurable (at build time): buffer sizes, number of buffers, maximum number of endpoints
- API calls are simple send / receive between endpoints
- Received data can be passed by a pointer or it can be copied
- Copy-less message sending mechanism
- Variable timeouts

2 Design Overview

2.1 Endpoints

User applications communicate by sending data to endpoints.

Endpoints are receive buffer queues, implemented in shared RAM, and are addressed by a triplet containing core, node, and port:

- **Core** - Identifies the core within the processor. For the Vybrid processor, the Cortex-A5 is core 0, and the Cortex-M4 is core 1.
- **Node** - In Linux, any user process participating in MCC is a unique node. Node numbering is arbitrary. MQX has only one node and it can also be an arbitrary number.
- **Port** - Both Linux and MQX can have an arbitrary number of ports per node up to a configurable maximum, arbitrarily numbered, with the exception of the port 0 (MCC_RESERVED_PORT_NUMBER), which is reserved.

The [core, node, port] triplets are part of the user's system design and implementation. There is no discovery protocol.

2.2 Buffer Management

Data is transferred between cores in fixed size buffers, allocated in shared RAM.

All buffers are allocated at initialization time in the free buffer pool. There is only one free buffer pool that is shared by all cores for data transfers in both directions.

When sending, data is always copied from the user application into the buffer. If no buffer is available at the time of the send call, the user application can wait for a specified amount of time, wait "forever", or have an error condition returned.

When receiving, the API supports receiving with copying to a user supplied buffer, or the application can request a pointer to the buffer and, therefore, eliminating the copy. In the latter case, the user application is responsible for freeing the buffer by the appropriate API call. Similarly to sending data, the user application can choose whether or not to wait for a received message. The user applications communicate by sending data to each other's endpoints.

2.3 Shared RAM

This is used as "bookkeeping data", such as endpoint and signal queue head and tail pointers, and fixed size data buffers. For the Vybrid processor, 64 KB of shared SRAM is used for MCC by default.

Each of the following can be configured at build time in *mcc_config.h* file. The numbers in parenthesis designate default values:

- Maximum number of endpoints (5)
- Number of data buffers (10)
- Size of each data buffer in bytes (1024)

All accesses to shared RAM are protected by hardware semaphores.

2.4 Signaling

The MCC running on one core interrupts the other core when:

- A buffer has been queued to an endpoint on the interrupted core.
- A buffer has been freed by the interrupting core.

There is one signal queue per core. A signal queue indicates the type of the interrupt, whether it is a queued or a free interrupt. If the interrupt is queued, a signal queue indicates the endpoint.

Each of the following can be configured at build time in mcc_config.h file. The numbers in parenthesis are default values:

- Maximum number of outstanding signals (10)

2.5 No-copy send/recv mechanisms

The MCC library implements no-copy mechanisms for both sending and receiving operations. These methods have their specifics that have to be considered when used in an application.

no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the MCC buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- call the **mcc_get_buffer()** API function to dequeue an MCC buffer from the free list and provide the MCC buffer pointer to the application
- fill the data to be sent into the pre-allocated MCC buffer - ensure that the filled data does not exceed the buffer size (provided as the mcc_get_buffer() return value)
- call the **mcc_send_nocopy()** API function to send the message to the destination endpoint - consider the cache functionality and the MCC buffer alignment, see the mcc_send_nocopy() function description below!

no-copy-receive mechanism: This mechanism allows reading messages without the cost for copying data from the MCC buffer in the shared memory to the application buffer. The sequence of no-copy receiving steps to be performed is as follows:

- call the **mcc_recv_nocopy()** API function to get the MCC buffer pointer to the received data
- read received data directly from the shared memory
- call the **mcc_free_buffer()** API function to release the MCC buffer and to make it available for the next data transfer

All no-copy mechanisms (API functions) are disabled in the mcc_config.h by default in order not to cause unexpected isssue when improperly used in an application. Once the user needs this functionality and is aware of all specifics and limitations of this approach it can be enabled by switching the **MCC_SEND_RECV_NOCOPY_API_ENABLED** macro to (1) in mcc_config.h.

2.6 Version Control

The **mcc_get_info()** API call returns a version string in the form of **mmm.nnn** where **mmm** is the major number of the version and **nnn** is the minor number. The major number of MQX and Linux **must match** to ensure compatibility. The minor numbers may be different.

Starting with MCC 2.0 version, the check for the version compatibility is done during the MCC library initialization phase when the **mcc_initialize()** API function is called, see **mcc_initialize()** return values.

3 MCC API Reference

MCC library API functions are located in the *mcc_api.c* file. These functions have the "mcc_" prefix and are listed in this chapter.

3.1 Function Listing Format

This is the general format for listing a function or a data type.

function_name()

A brief description of what function **function_name()** does.

Prototype

Provides a prototype for the function **function_name()**.

```
<return_type> function_name(<type_1> parameter_1,<type_2> parameter_2, ... <type_n>
parameter_n)
```

Parameters

Function parameters are listed in tables. See an example table below.

Type	Name	Direction	Description
void *	buffer	**input**	Pointer to the user-app. buffer data will be copied into.

- **Type:** Parameter data type.
- **Name:** Parameter name.
- **Direction:**
  - input - It means the function uses one or more values in the parameter you give it without storing any changes.
  - output - It means the function saves one or more values in the parameter you give it. You can view the saved values to find out useful information about your application.
  - input, output - It means the function changes one or more values in the parameter you give it and saves the result. You can view the saved values to find out useful information about your application.
- **Description:** Description for each parameter.

Returns

Specifies any value or values returned by a function.

See also

Lists other functions or data types related to the function **function_name()**.

Example

Provides an example, or a reference to an example, that illustrates the use of function **function_name()**.

Description

Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- The function blocks, or might block under certain conditions.
- The function must be started as a task.
- The function creates a task.
- The function has pre-conditions that might not be obvious.
- The function has restrictions or special behavior.

3.2 mcc_create_endpoint

This function creates an endpoint.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_create_endpoint(MCC_ENDPOINT *endpoint, MCC_PORT port);`

Table 1. mcc_create_endpoint arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	endpoint	**output**	Pointer to the endpoint triplet to be created.
MCC_PORT	port	**input**	Port number.

Returns :

- MCC_SUCCESS
- MCC_ERR_NOMEM (maximum number of endpoints exceeded)
- MCC_ERR_ENDPOINT (invalid value for port or endpoint already registered)
- MCC_ERR_SEMAPHORE (semaphore handling error)

See also :

- mcc_destroy_endpoint
- MCC_ENDPOINT

Description :

The function creates an endpoint on the local node with the specified port number. The core and node provided in the endpoint must match the caller's core and node, and the port argument must match the endpoint port.

3.3 mcc_destroy

This function de-initializes the Multi Core Communication subsystem for a given node.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_destroy(MCC_NODE node);`

Table 2. mcc_destroy arguments

Type	Name	Direction	Description
MCC_NODE	node	**input**	Node number to be deinitialized.

Returns :

- MCC_SUCCESS
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_OSSYNC (OS synchronization module(s) deinitialization failed)

See also :

- mcc_initialize

Description :

The function frees all resources of the node. Deletes all endpoints and frees any buffers that may have been queued there.

3.4 mcc_destroy_endpoint

This function destroys an endpoint.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_destroy_endpoint(MCC_ENDPOINT *endpoint);`

Table 3. mcc_destroy_endpoint arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	endpoint	**input**	Pointer to the endpoint triplet to be deleted.

Returns :

- MCC_SUCCESS
- MCC_ERR_ENDPOINT (the endpoint doesn't exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)

See also :

- mcc_create_endpoint
- MCC_ENDPOINT

Description :

The function destroys an endpoint on the local node and frees any buffers that may be queued.

3.5 mcc_free_buffer

This function frees a buffer previously returned by mcc_recv_nocopy.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_free_buffer(void *buffer);`

Table 4. mcc_free_buffer arguments

Type	Name	Direction	Description
void *	buffer	**input**	Pointer to the buffer to be freed.

Returns :

- MCC_SUCCESS
- MCC_ERR_SEMAPHORE (semaphore handling error)

See also :

- mcc_recv_nocopy

Description :

Once the zero-copy mechanism of receiving data is used, this function has to be called to free a buffer and to make it available for the next data transfer.

3.6 mcc_get_buffer

This function dequeues a buffer from the free list.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_get_buffer(void **buffer, MCC_MEM_SIZE *buf_size, unsigned int timeout_ms);`

Table 5. mcc_get_buffer arguments

Type	Name	Direction	Description
void **	buffer	**output**	Pointer to the MCC buffer dequeued from the free list.
MCC_MEM_SIZE *	buf_size	**output**	Pointer to an MCC_MEM_SIZE that is used for passing the size of the dequeued MCC buffer to the application.

Table continues on the next page...

Table 5. mcc_get_buffer arguments (continued)

Type	Name	Direction	Description
unsigned int	timeout_ms	**input**	Timeout, in milliseconds, to wait for a free buffer. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns :
- MCC_SUCCESS
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_TIMEOUT (timeout exceeded before a buffer became available)
- MCC_ERR_NOMEM (no free buffer available and timeout_ms set to 0)

See also :
- mcc_send_nocopy

Description :

The application has take the responsibility for MCC buffer de-allocation and filling the data to be sent into the pre-allocated MCC buffer.

3.7 mcc_get_info

This function returns information about the MCC sub system.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_get_info(MCC_NODE node, MCC_INFO_STRUCT *info_data);`

Table 6. mcc_get_info arguments

Type	Name	Direction	Description
MCC_NODE	node	**input**	Node number.
MCC_INFO_STRUCT *	info_data	**output**	Pointer to the MCC_INFO_STRUCT structure to hold returned data.

Returns :
- MCC_SUCCESS
- MCC_ERR_SEMAPHORE (semaphore handling error)

See also :
- MCC_INFO_STRUCT

Description :

The function returns implementation-specific information.

3.8 mcc_initialize

This function initializes the Multi Core Communication subsystem for a given node.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_initialize(MCC_NODE node);`

Table 7. mcc_initialize arguments

Type	Name	Direction	Description
MCC_NODE	node	**input**	Node number that will be used in endpoints created by this process.

Returns :
- MCC_SUCCESS
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_INT (interrupt registration error)
- MCC_ERR_VERSION (incorrect MCC version used - compatibility issue)
- MCC_ERR_OSSYNC (OS synchronization module(s) initialization failed)

See also :
- mcc_destroy
- MCC_BOOKEEPING_STRUCT

Description :

This function should only be called once per node (once in MQX, once per a process in Linux). It tries to initialize the bookkeeping structure when the init_string member of this structure is not equal to MCC_INIT_STRING, i.e. when no other core had performed the initialization yet. Note, that this way of bookkeeping data re-initialization protection is not powerful enough and the user application should not rely on this method. Instead, the application should be designed to unambiguously assign the core that will perform the MCC initialization. Clear the shared memory before the first core is attempting to initialize the MCC (in some cases MCC_INIT_STRING remains in the shared memory after the application reset and could cause that the bookkeeping data structure is not initialized correctly).

3.9 mcc_msgs_available

This function returns the number of buffers currently queued at the endpoint.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_msgs_available(MCC_ENDPOINT *endpoint, unsigned int *num_msgs);`

Table 8. mcc_msgs_available arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	endpoint	**input**	Pointer to the endpoint structure.
unsigned int *	num_msgs	**output**	Pointer to an unsigned int that will contain the number of buffers queued.

Returns :
- MCC_SUCCESS
- MCC_ERR_ENDPOINT (the endpoint does not exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)

See also :
- mcc_recv
- mcc_recv_nocopy
- MCC_ENDPOINT

Description :

The function checks if messages are available on a receive endpoint. While the call only checks the availability of messages, it does not dequeue them.

3.10 mcc_recv

This function receives a message from the specified endpoint if one is available. The data is copied from the receive buffer into the user supplied buffer.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_recv(MCC_ENDPOINT *src_endpoint, MCC_ENDPOINT *dest_endpoint, void *buffer, MCC_MEM_SIZE buffer_size, MCC_MEM_SIZE *recv_size, unsigned int timeout_ms);`

Table 9. mcc_recv arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	src_endpoint	**output**	Pointer to the MCC_ENDPOINT structure to be filled by the endpoint identifying the message sender.
MCC_ENDPOINT *	dest_endpoint	**input**	Pointer to the local receiving endpoint to receive from.
void *	buffer	**input**	Pointer to the user-app. buffer where data will be copied into.
MCC_MEM_SIZE	buffer_size	**input**	The maximum number of bytes to copy.
MCC_MEM_SIZE *	recv_size	**output**	Pointer to an MCC_MEM_SIZE that will contain the number of bytes actually copied into the buffer.
unsigned int	timeout_ms	**input**	Timeout, in milliseconds, to wait for a free buffer. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns :

- MCC_SUCCESS
- MCC_ERR_ENDPOINT (the endpoint does not exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_TIMEOUT (timeout exceeded before a new message came)

See also :

- mcc_send
- mcc_recv_nocopy
- MCC_ENDPOINT

Description :

This is the "receive with copy" version of the MCC receive function. This version is simple to use but it requires copying data from shared memory into the user space buffer. The user has no obligation or burden to manage the shared memory buffers.

3.11 mcc_recv_nocopy

This function receives a message from the specified endpoint if one is available. The data is NOT copied into the user-app. buffer.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_recv_nocopy(MCC_ENDPOINT *src_endpoint, MCC_ENDPOINT *dest_endpoint, void **buffer_p, MCC_MEM_SIZE *recv_size, unsigned int timeout_ms);`

Table 10. mcc_recv_nocopy arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	src_endpoint	**output**	Pointer to the MCC_ENDPOINT structure to be filled by the endpoint identifying the message sender.
MCC_ENDPOINT *	dest_endpoint	**input**	Pointer to the local receiving endpoint to receive from.
void **	buffer_p	**output**	Pointer to the MCC buffer of the shared memory where the received data is stored.
MCC_MEM_SIZE *	recv_size	**output**	Pointer to an MCC_MEM_SIZE that will contain the number of valid bytes in the buffer.
unsigned int	timeout_ms	**input**	Timeout, in milliseconds, to wait for a free buffer. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns :
- MCC_SUCCESS
- MCC_ERR_ENDPOINT (the endpoint does not exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_TIMEOUT (timeout exceeded before a new message came)

See also :
- mcc_free_buffer
- mcc_send
- mcc_recv
- MCC_ENDPOINT

Description :

This is the "zero-copy receive" version of the MCC receive function. No data is copied. Only the pointer to the data is returned. This version is fast, but it requires the user to manage buffer allocation. Specifically, the user must decide when a buffer is no longer in use and make the appropriate API call to free it, see mcc_free_buffer.

3.12 mcc_send

This function sends a message to an endpoint.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_send(MCC_ENDPOINT *src_endpoint, MCC_ENDPOINT *dest_endpoint, void *msg, MCC_MEM_SIZE msg_size, unsigned int timeout_ms);`

Table 11. mcc_send arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	src_endpoint	**input**	Pointer to the local endpoint identifying the source endpoint.
MCC_ENDPOINT *	dest_endpoint	**input**	Pointer to the destination endpoint to send the message to.
void *	msg	**input**	Pointer to the message to be sent.
MCC_MEM_SIZE	msg_size	**input**	Size of the message to be sent in bytes.

Table continues on the next page...

Table 11. mcc_send arguments (continued)

Type	Name	Direction	Description
unsigned int	timeout_ms	**input**	Timeout, in milliseconds, to wait for a free buffer. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns :
- MCC_SUCCESS
- MCC_ERR_ENDPOINT (the endpoint does not exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_INVAL (the msg_size exceeds the size of a data buffer)
- MCC_ERR_TIMEOUT (timeout exceeded before a buffer became available)
- MCC_ERR_NOMEM (no free buffer available and timeout_ms set to 0)
- MCC_ERR_SQ_FULL (signal queue is full)

See also :
- mcc_recv
- mcc_recv_nocopy
- MCC_ENDPOINT

Description :

The message is copied into the MCC buffer and the destination core is signaled.

3.13 mcc_send_nocopy

This function sends a message to an endpoint. The data is NOT copied from the user-app. buffer but the pointer to already filled message buffer is provided.

Source : `/mcc/source/mcc_api.c`

Prototype : `int mcc_send_nocopy(MCC_ENDPOINT *src_endpoint, MCC_ENDPOINT *dest_endpoint, void *buffer_p, MCC_MEM_SIZE msg_size);`

Table 12. mcc_send_nocopy arguments

Type	Name	Direction	Description
MCC_ENDPOINT *	src_endpoint	**input**	Pointer to the local endpoint identifying the source endpoint.
MCC_ENDPOINT *	dest_endpoint	**input**	Pointer to the destination endpoint to send the message to.
void *	buffer_p	**input**	Pointer to the MCC buffer of the shared memory where the data to be sent is stored.
MCC_MEM_SIZE	msg_size	**input**	Size of the message to be sent in bytes.

Returns :
- MCC_SUCCESS
- MCC_ERR_INVAL (the msg_size exceeds the size of a data buffer)
- MCC_ERR_ENDPOINT (the endpoint does not exist)
- MCC_ERR_SEMAPHORE (semaphore handling error)
- MCC_ERR_SQ_FULL (signal queue is full)

See also :

- mcc_send
- mcc_get_buffer
- MCC_ENDPOINT

Description :

The application has to take the responsibility for:

1. MCC buffer de-allocation
2. filling the data to be sent into the pre-allocated MCC buffer
3. not exceeding the buffer size when filling the data (MCC_ATTR_BUFFER_SIZE_IN_BYTES)

Once the data cache is used on the target platform it is good to have MCC buffers in shared RAM aligned to the cache line size in order not to corrupt entities placed just before and just after the MCC buffer when flushing the MCC buffer content into the shared RAM. It is also the application responsibility to flush the data in that case. If the alignment condition is not fulfilled the application has to take care about the data cache coherency.

The following scenarios can happen:

A. Data cache is OFF:

- No cache operation needs to be done, the application just
  a. calls the mcc_get_buffer function,
  b. fills data into the provided MCC buffer,
  c. and finally issues the mcc_send_nocopy function.

B. Data cache is ON, shared RAM MCC buffers ALIGNED to the cache line size:

- The application has to perform following steps:
  a. call the mcc_get_buffer to get the pointer to a free message buffer
  b. copy data to be sent into the message buffer
  c. flush all cache lines occupied by the message buffer new data (maximum of MCC_ATTR_BUFFER_SIZE_IN_BYTES bytes).
  d. call the mcc_send_nocopy with the correct buffer pointer and the message size passed

C. Data cache is ON, shared RAM MCC buffers NOT ALIGNED:

- The application has to perform following steps:
  a. call the mcc_get_buffer to get the pointer to a free message buffer
  b. grab the hw semaphore by calling the mcc_get_semaphore() low level MCC function.
  c. invalidate all cache lines occupied by data to be filled into the free message buffer. (maximum of MCC_ATTR_BUFFER_SIZE_IN_BYTES bytes).
  d. copy data to be sent into the message buffer.
  e. flush all cache lines occupied by the message buffer new data (maximum of MCC_ATTR_BUFFER_SIZE_IN_BYTES bytes).
  f. release the hw semaphore by calling the mcc_release_semaphore() low level MCC function.
  g. call the mcc_send_nocopy with the correct buffer pointer and the message size passed.

After the mcc_send_nocopy function is issued the message buffer is no more owned by the sending task and must not be touched anymore unless the mcc_send_nocopy function fails and returns an error. In that case the application should try to re-issue the mcc_send_nocopy again and if it is still not possible to send the message and the application wants to give it up from whatever reasons (for instance the MCC_ERR_ENDPOINT error is returned meaning the endpoint has not been created yet) the mcc_free_buffer function could be called, passing the pointer to the buffer to be freed as a parameter.

4 MCC Data Types

4.1 MCC_BOOKEEPING_STRUCT

Share Memory data - Bookkeeping data and buffers.

Description :

This is used for "bookkeeping data" such as endpoint and signal queue head and tail pointers and fixed size data buffers. The whole mcc_bookeeping_struct as well as each individual structure members has to be defined and stored in the memory as packed structure. This way, the same structure member offsets will be ensured on all cores/OSes/compilers. Compiler-specific pragmas for data packing have to be applied.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  char                   init_string[sizeof(MCC_INIT_STRING)],
  char                   version_string[sizeof(MCC_VERSION_STRING)],
  MCC_RECEIVE_LIST       free_list,
  MCC_SIGNAL             signals_received[MCC_NUM_CORES][MCC_MAX_OUTSTANDING_SIGNALS],
  unsigned int           signal_queue_head[MCC_NUM_CORES],
  unsigned int           signal_queue_tail[MCC_NUM_CORES],
  MCC_ENDPOINT_MAP_ITEM endpoint_table[MCC_ATTR_MAX_RECEIVE_ENDPOINTS],
  MCC_RECEIVE_BUFFER     r_buffers[MCC_ATTR_NUM_RECEIVE_BUFFERS]
} MCC_BOOKEEPING_STRUCT;
```

See also :

- MCC_RECEIVE_LIST
- MCC_SIGNAL
- MCC_ENDPOINT_MAP_ITEM
- MCC_RECEIVE_BUFFER

Table 13. Structure MCC_BOOKEEPING_STRUCT member description

Member	Description
init_string	String that indicates if this structure has been already initialized.
version_string	String that indicates the MCC library version.
free_list	List of free buffers.
signals_received	Each core has it's own queue of received signals.
signal_queue_head	Signal queue head for each core.
signal_queue_tail	Signal queue tail for each core.
endpoint_table	Endpoint map.
r_buffers	Receive buffers, the number is defined in mcc_config.h (MCC_ATTR_NUM_RECEIVE_BUFFERS)

4.2 MCC_ENDPOINT

Endpoint structure.

Description :

Endpoints are receive buffer queues, implemented in shared RAM, and are addressed by a triplet containing core, node, and port.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  MCC_CORE core,
  MCC_NODE node,
  MCC_PORT port
} MCC_ENDPOINT;
```

See also :

- MCC_BOOKEEPING_STRUCT

Table 14. Structure MCC_ENDPOINT member description

Member	Description
core	Core number - identifies the core within the processor.
node	Node number - in Linux any user process participating in MCC is a unique node; MQX has only one node.
port	Port number - both Linux and MQX can have an arbitrary number of ports per node.

4.3 MCC_ENDPOINT_MAP_ITEM

Endpoint registration table.

Description :

This is used for matching each endpoint structure with it's list of received buffers.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  MCC_ENDPOINT     endpoint,
  MCC_RECEIVE_LIST list
} MCC_ENDPOINT_MAP_ITEM;
```

See also :

- MCC_ENDPOINT
- MCC_RECEIVE_LIST
- MCC_BOOKEEPING_STRUCT

Table 15. Structure MCC_ENDPOINT_MAP_ITEM member description

Member	Description
endpoint	Endpoint tripplet.
list	List of received buffers.

4.4 MCC_INFO_STRUCT

MCC info structure.

Description :

This is used for additional information about the MCC implementation.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  char version_string[sizeof(MCC_VERSION_STRING)]
} MCC_INFO_STRUCT;
```

See also :
- MCC_BOOKEEPING_STRUCT

Table 16. Structure MCC_INFO_STRUCT member description

Member	Description
version_string	<major>.<minor> - minor is changed whenever patched, major indicates compatibility

4.5 MCC_RECEIVE_BUFFER

Receive buffer structure.

Description :

This is the receive buffer structure used for exchanging data.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  struct mcc_receive_buffer * next,
  MCC_ENDPOINT                source,
  MCC_MEM_SIZE                data_len,
  char                        data[MCC_ATTR_BUFFER_SIZE_IN_BYTES]
} MCC_RECEIVE_BUFFER;
```

See also :
- MCC_BOOKEEPING_STRUCT

Table 17. Structure MCC_RECEIVE_BUFFER member description

Member	Description
next	Pointer to the next receive buffer.
source	Source endpoint.

Table continues on the next page...

Table 17. Structure MCC_RECEIVE_BUFFER member description (continued)

Member	Description
data_len	Length of data stored in this buffer.
data	Space for data storage.

4.6 MCC_RECEIVE_LIST

List of buffers.

Description :

Each endpoint keeps the list of received buffers. The list of free buffers is kept in bookkeeping data structure.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  MCC_RECEIVE_BUFFER * head,
  MCC_RECEIVE_BUFFER * tail
} MCC_RECEIVE_LIST;
```

See also :
- MCC_RECEIVE_BUFFER
- MCC_BOOKEEPING_STRUCT

Table 18. Structure MCC_RECEIVE_LIST member description

Member	Description
head	Head of a buffers list.
tail	Tail of a buffers list.

4.7 MCC_SIGNAL

Signals and signal queues.

Description :

This is one item of a signal queue.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef struct
{
  MCC_SIGNAL_TYPE type,
  MCC_ENDPOINT    destination
} MCC_SIGNAL;
```

See also :

- MCC_ENDPOINT
- MCC_BOOKEEPING_STRUCT

Table 19. Structure MCC_SIGNAL member description

Member	Description
type	Signal type - BUFFER_QUEUED or BUFFER_FREED.
destination	Destination endpoint.

4.8 MCC_MEM_SIZE

MCC_MEM_SIZE type.

Description :

Mem size type definiton for the MCC library.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef unsigned int MCC_MEM_SIZE
```

4.9 MCC_CORE

MCC_CORE type.

Description :

This unsigned integer value specifies the core number for the endpoint definition.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef unsigned int MCC_CORE
```

See also :

- MCC_ENDPOINT

4.10 MCC_NODE

MCC_NODE type.

Description :

This unsigned integer value specifies the node number for the endpoint definition.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef unsigned int MCC_NODE
```

See also :

- MCC_ENDPOINT

4.11 MCC_PORT

MCC_PORT type.

Description :

This unsigned integer value specifies the port number for the endpoint definition.

Source : `/mcc/source/include/mcc_common.h`

Declaration :

```
typedef unsigned int MCC_PORT
```

See also :

- MCC_ENDPOINT

5 API Example

As part of the system design, the designers agreed that MQX would be node 0 and the application would be receiving on port 2.

Therefore, Linux will send to [1,0,2]. Similarly, Linux will receive on [0,0,1].

Linux pseudo code:

```
      mcc_initialize(0)
      mcc_create_endpoint([0,0,1], 1)
      mcc_send([0,0,1], [1,0,2], "hello", 5, 50) // no more than 50 milliseconds for buffer
      mcc_recv_copy(&src_ep, [0,0,1], &buf, sizeof(buf), length, 0xffffffff) //forever
```

MQX pseudo code:

```
      mcc_initialize(0)
      mcc_create_endpoint([1,0,2], 2)
      mcc_recv_nocopy(&src_ep, [1,0,2], &buf_p, length, 0xffffffff) //forever
      mcc_send([1,0,2], [0,0,1], "hello", 5, 50) // no more than 50 milliseconds for buffer
```

MCC example applications are located in the following MQX installation directory:

<mqx_install_dir>/mcc/examples

Refer to the general MQX documentation (*<mqx_install_dir>/doc/mqx*) and to the tools-specific MQX documentation (*<mqx_install_dir>/doc/tools*) for instructions about how to build and run these applications. See also the readme file attached to each MCC example to get instructions specific to the particular example application.

6 Version History

Table 20. Version history table

Revision	Date	Features
1.0	Apr 2013	Initial version.
1.1	Aug 2013	Wrong usage of cache macros in several functions fixed. MCC_VERSION_STRING updated to 1.1. mcc_recv_nocopy and mcc_initialize function descriptions updated. No changes in API introduced.
1.2	Dec 2013	Protection of the signal queue when accessing from the cpu-to-cpu isr added. No changes in API introduced.
2.0	Sep 2014	This release breaks the backward compatibility (API changes) and brings the following new features: • no-copy-send mechanism implemented • sender endpoint propagation to the receiver side implemented • OS-abstraction code consolidated • timeout definition changed from microseconds to milliseconds
2.1	Apr 2015	This release fixes following issues: • the way mcc_recv_common_part() function is waiting for the a new message • the order of data cache invalidation in mcc_recv_nocopy() function (not correct message payload received) • minor arrangements of data cache macros in MCC common part MCC_VERSION_STRING updated to 2.1. No changes in API introduced.

Document Number MQXMCCUG
Revision 2.1