



Nucleus® IPC Guide

Release 2013.08

August 2013

**© 2010-2013 Mentor Graphics Corporation
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1	
Nucleus IPC Component	7
Operating System Support	7
Build Configurations	7
Required Include File	7
Chapter 2	
MCAPI Overview	9
Topology	9
Nodes	9
Endpoints	10
Sending and Receiving Data	11
Messages	12
Packet Channels	12
Scalar Channels	14
Blocking / Non-Blocking Functions	14
Chapter 3	
Configuring MGC MCAPI	17
Topology Configuration	17
Interface Configuration	17
Route Configuration	17
Compile-Time Macros	17
MCAPI_BUF_COUNT	17
MCAPI_DEFAULT_PRIO	17
MCAPI_ENABLE_FORWARD	18
MCAPI_ENABLE_LOOPBACK	18
MCAPI_ENDPOINT_PORT_INIT	18
MCAPI_FREE_REQUEST_COUNT	18
MCAPI_INT_NAME_LEN	18
MCAPI_INTERFACE_COUNT	18
MCAPI_MAX_DATA_LEN	18
MCAPI_MAX_ENDPOINTS	19
MCAPI_PRIO_COUNT	19
MCAPI_ROUTE_COUNT	19
MCAPI_RX_CONTROL_PORT	19
Chapter 4	
Function Reference	21
Endpoint Primitives	22
mcati_create_endpoint	23
mcati_delete_endpoint	25

mcapi_get_endpoint	27
mcapi_get_endpoint_attribute	29
mcapi_set_endpoint_attribute	31
mcapi_get_endpoint_i	33
Message Primitives	35
mcapi_msg_available	36
mcapi_msg_recv	38
mcapi_msg_recv_i	40
mcapi_msg_send	42
mcapi_msg_send_i	44
Node Primitives	47
mcapi_initialize	48
mcapi_finalize	50
mcapi_get_node_id	51
Packet Channel Primitives	52
mcapi_connect_pktchan_i	53
mcapi_open_pktchan_recv_i	55
mcapi_open_pktchan_send_i	57
mcapi_packetchan_recv_close_i	59
mcapi_packetchan_send_close_i	61
mcapi_pktchan_available	63
mcapi_pktchan_free	65
mcapi_pktchan_recv	67
mcapi_pktchan_recv_i	69
mcapi_pktchan_send	71
mcapi_pktchan_send_i	73
Primitives to Manage Non-Blocking Operations	75
mcapi_cancel	76
mcapi_test	77
mcapi_wait	79
mcapi_wait_any	81
Scalar Channel Primitives	83
mcapi_connect_sclchan_i	84
mcapi_open_sclchan_recv_i	86
mcapi_open_sclchan_send_i	88
mcapi_sclchan_available	90
mcapi_sclchan_recv_close_i	92
mcapi_sclchan_recv_uint8	94
mcapi_sclchan_recv_uint16	96
mcapi_sclchan_recv_uint32	98
mcapi_sclchan_recv_uint64	100
mcapi_sclchan_send_close_i	102
mcapi_sclchan_send_uint8	104
mcapi_sclchan_send_uint16	106
mcapi_sclchan_send_uint32	108
mcapi_sclchan_send_uint64	110

Chapter 5

Porting MGC MCAPI	113
OS-Specific Directory Structure	113
Porting the Data Structures	113
MCAPI_POINTER	113
mcapi_cond_t	114
Initialization and Shut Down	114
MCAPI_Init_OS	114
MCAPI_Cleanup_Task	114
MCAPI_Exit_OS	114
Protecting Data	114
MCAPI_Create_Mutex	114
MCAPI_Delete_Mutex	114
MCAPI_Obtain_Mutex	115
MCAPI_Release_Mutex	115
Delivering Data to MCAPI	115
MCAPI_Set_RX_Event	117
MCAPI_Lock_RX_Queue	117
MCAPI_Unlock_RX_Queue	117
Suspending / Resuming Threads	117
MCAPI_Init_Condition	118
MCAPI_Set_Condition	118
MCAPI_Clear_Condition	118
MCAPI_Suspend_Task	118
MCAPI_Resume_Task	119

Embedded Software and Hardware License Agreement

List of Figures

Figure 2-1. Data Flow between Nodes 10

Figure 2-2. Data Flow Using Endpoints 11

Figure 2-3. Data Flow Using Connectionless Endpoints..... 12

Figure 2-4. Data Flow Using Two Connections 13

Figure 5-1. Nucleus Data Delivery and MCAPi 116

Figure 5-2. Linux Data Delivery and MCAPi..... 116

Chapter 1

Nucleus IPC Component

The Nucleus IPC component is an implementation for inter-core communication within closely distributed systems (called MCAPI in this guide). The current implementation is based on version 1.063 of the Multicore Communications API published by the Multicore Association.

This component is part of the Kernel package and it is located at `\os\kernel\ipc\mcapi`. Depending on your embedded application, any of these components can be configured for use.

Note



Your source code is initially located in the `<install_root>\nucleus` directory. The source from this directory is copied into the project folder you specify.

This guide describes in detail the MCAPI APIs for application usage. For more information about packages and components, see “Nucleus ReadyStart Configuration” in the *Nucleus ReadyStart Guide* or “Nucleus Source Code Configuration” in the *Nucleus Source Code Guide*.

Operating System Support

The MGC MCAPI implementation provides support for the Nucleus PLUS and Linux operating systems. Additional OS support can be enabled by porting the MCAPI porting layer routines to use the new operating system.

Build Configurations


All components are by default enabled through the `.metadata` file located at the top level of each component’s installation, for example for MCAPI: `\os\kernel\ipc`. When a component is enabled, it is included in the build. You can create a user configuration file to override the default configuration and exclude a component from the build.

For more information, see “Creating a Custom Configuration” in the *Nucleus ReadyStart Guide* or in the *Nucleus Source Code Guide*.

Required Include File

To make the MCAPI Component APIs visible to an application, include the following files in your application using the following statements:

```
#include "mcapi/openmcapi_cfg.h"  
#include "mcapi/mcapi.h"  
#include "mcapi/linux/mcapi_os.h"  
#include "mcapi/nucleus/mcapi_os.h"
```

 **Warning** In your applications, use only interfaces, structures, macros, and so on, that are documented within this and other Nucleus guides. There is no guarantee of future support or compatibility for any interface that is not documented.

Chapter 2

MCAPI Overview

This section describes the components of an MCAPI system, explaining the MCAPI topology, how data is sent and received, and how blocking and non-blocking functions are used.

Topology

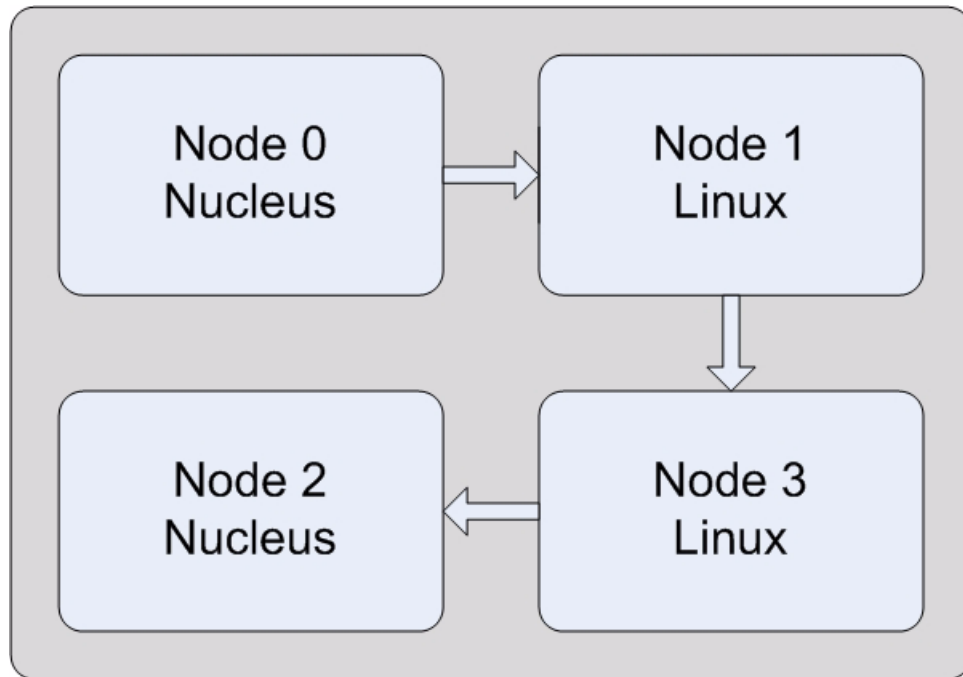
The MCAPI topology consists of independent nodes exchanging data via endpoints. Nodes are pre-determined at compile-time, whereas endpoints can be configured at compile-time or run-time.

Nodes

Within the MGC implementation of MCAPI, a node is a single instance of an operating system. Nodes must be defined at compile-time and cannot change during run-time. Each node is referenced by a system-wide unique Node ID configured by the user at initialization. A node is considered reachable by another node if a route to the foreign node exists on the local node. If two nodes do not need to exchange data, there is no reason for those nodes to be reachable to each other in the system.

Figure 2-1 shows a simplistic example of a node with Node ID '0' sending data to the node with Node ID '1,' and each successive node passes the data on to the following node. The arrows represent the flow of data within the system and the required routes configured on the respective nodes. Because Node 2 is the termination point of the data, it has no arrow and therefore no route to any node.

Figure 2-1. Data Flow between Nodes



MCAPI Node Primitives

The following MCAPI primitives are used to operate on nodes:

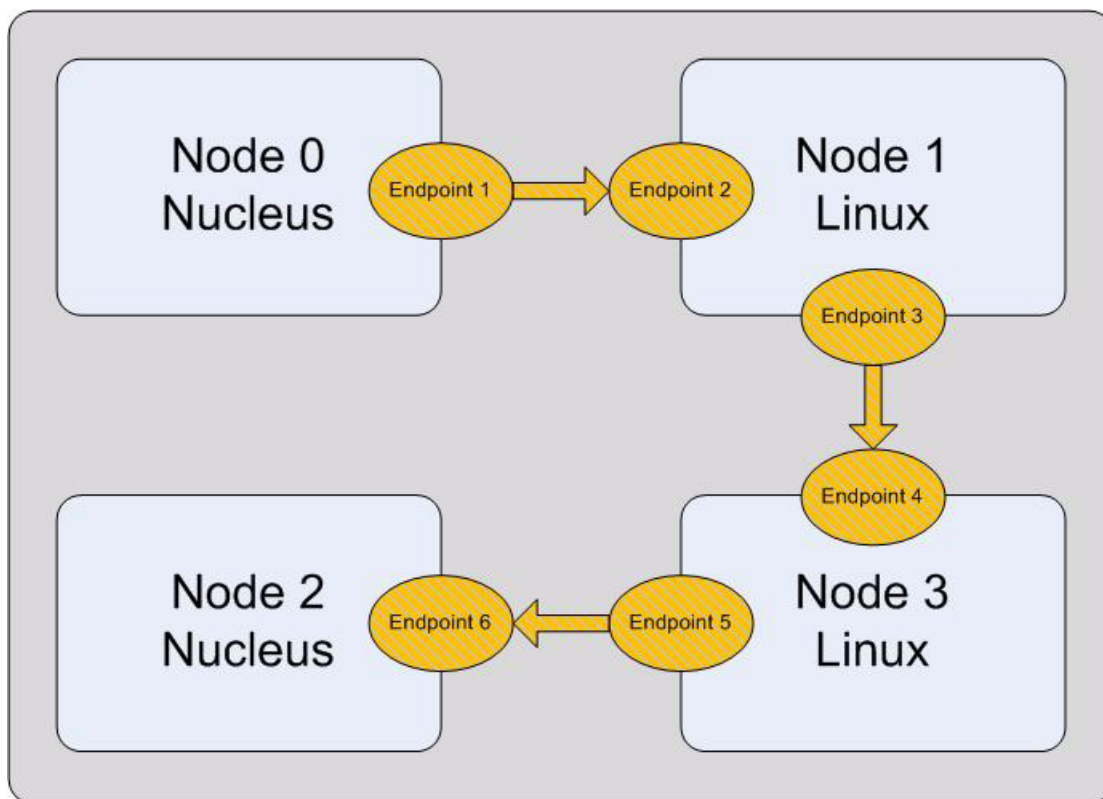
- [mcapi_initialize](#)
- [mcapi_finalize](#)
- [mcapi_get_node_id](#)

Endpoints

An endpoint is a logical communication point used to send and receive data across nodes in a system. Their usage is similar to that of TCP/IP sockets; the application creates an endpoint and uses MCAPI primitives to issue data transmission calls.

Extending the example above, [Figure 2-2](#) shows Node 0 and Node 1 communicate using endpoints 1 and 2. Node 1 and Node 3 communicate using endpoints 3 and 4, continuing in the same pattern. Each node must know how to reach the respective foreign endpoint. This is accomplished through knowledge of the Node ID and the port ID of the respective endpoint. This information must either be configured at compile-time or communicated at run-time using a well-known port ID. Using the <Node ID, port ID> tuple system to define an endpoint ensures the endpoint is unique within the system.

Figure 2-2. Data Flow Using Endpoints



MCAPI Endpoint Primitives

The following MCAPI primitives are used to operate on endpoints:

- [mcapi_create_endpoint](#)
- [mcapi_delete_endpoint](#)
- [mcapi_get_endpoint](#)
- [mcapi_get_endpoint_attribute](#)
- [mcapi_get_endpoint_i](#)
- [mcapi_set_endpoint_attribute](#)

Sending and Receiving Data

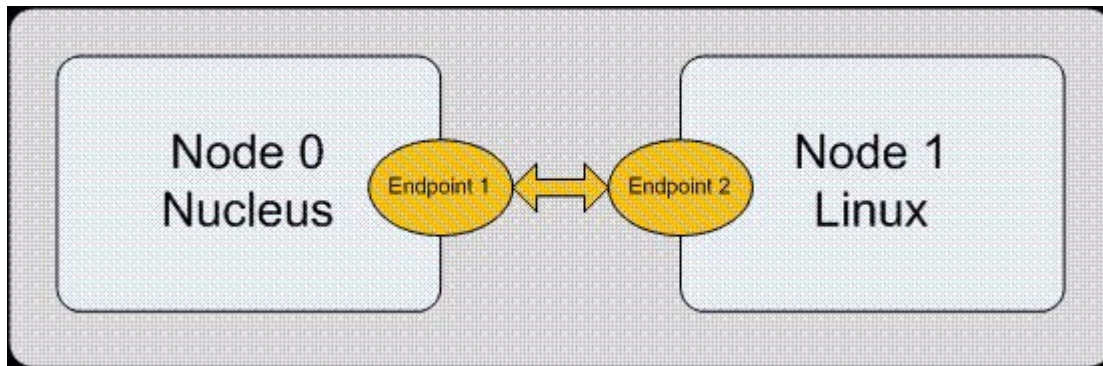
MCAPI provides three methods of sending and receiving data; connectionless [Messages](#), [Packet Channels](#) and [Scalar Channels](#).

Messages

MCAPI messages are used to send and receive data to and from any other endpoint in the system. Messages do not require negotiation of a connection and are the only type of transfer method that can be used bi-directionally; ie, an endpoint communicating using messages can send and receive data over the same endpoint. MCAPI messages are similar to UDP sockets in TCP/IP networking. There is currently no way to accomplish broadcast or multicast transmission using MCAPI primitives, but this functionality could be built on top of an MCAPI implementation at the application layer.

Figure 2-3 shows Node 0 and Node 1 communicating bi-directionally using connectionless messages over endpoints 1 and 2. This requires a route from Node 0 to Node 1 and a route from Node 1 to Node 0. Additionally, each node must know the target Node ID and port ID of the destination endpoint.

Figure 2-3. Data Flow Using Connectionless Endpoints



MCAPI Message Primitives

The following MCAPI primitives are used to send and receive connectionless messages:

- [mcapi_msg_available](#)
- [mcapi_msg_rcv](#)
- [mcapi_msg_rcv_i](#)
- [mcapi_msg_send](#)
- [mcapi_msg_send_i](#)

Packet Channels

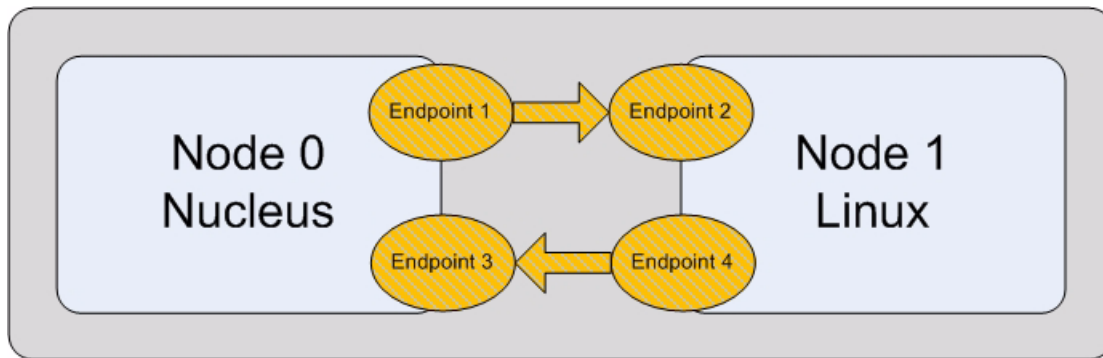
MCAPI packet channels are used to transmit data uni-directionally between a connected pair of endpoints. Before communication can take place, the send and receive endpoints must be connected via a three-way handshake. Packet channels are more restrictive than connectionless messages, but also more efficient since the destination is fixed.

Figure 2-4 shows Node 0 and Node 1 with two connections established; one between endpoints 1 and 2 and another between endpoints 3 and 4.

In the first connection, endpoint 1 is designated the sender and endpoint 2 the receiver. Node 0 can only send data over this connection, and Node 1 can only receive data.

The second connection illustrates the opposite scenario; Node 1 is the sender and Node 0 the receiver.

Figure 2-4. Data Flow Using Two Connections



MCAPI Packet Channel Primitives

The following MCAPI primitives are used to manage packet channels:

- [mcapi_connect_pktchan_i](#)
- [mcapi_open_pktchan_rcv_i](#)
- [mcapi_open_pktchan_send_i](#)
- [mcapi_packetchan_rcv_close_i](#)
- [mcapi_packetchan_send_close_i](#)
- [mcapi_pktchan_available](#)
- [mcapi_pktchan_free](#)
- [mcapi_pktchan_rcv](#)
- [mcapi_pktchan_rcv_i](#)
- [mcapi_pktchan_send](#)
- [mcapi_pktchan_send_i](#)

Scalar Channels

MCAPI scalar channels operate much like packet channels except that they are used to transmit scalar data. Just like with packet channels, before communication can take place over a scalar channel, the send and receive endpoints must be connected via a three-way handshake. Scalar channels can be used to transmit 8, 16, 32, or 64-bit scalars in a very efficient manner.

MCAPI Scalar Channel Primitives

The following MCAPI primitives are used to manage scalar channels:

- [mcapi_connect_sclchan_i](#)
- [mcapi_open_sclchan_recv_i](#)
- [mcapi_open_sclchan_send_i](#)
- [mcapi_sclchan_available](#)
- [mcapi_sclchan_recv_close_i](#)
- [mcapi_sclchan_recv_uint8](#)
- [mcapi_sclchan_recv_uint16](#)
- [mcapi_sclchan_recv_uint32](#)
- [mcapi_sclchan_recv_uint64](#)
- [mcapi_sclchan_send_close_i](#)
- [mcapi_sclchan_send_uint8](#)
- [mcapi_sclchan_send_uint16](#)
- [mcapi_sclchan_send_uint32](#)
- [mcapi_sclchan_send_uint64](#)

Blocking / Non-Blocking Functions

For many MCAPI primitives, there is both a blocking and non-blocking version of the function. The blocking version waits until the operation completes successfully, is canceled, or an error occurs. The non-blocking version (identified by the '_i' appended to the name) returns immediately.

You must query MCAPI to determine whether the operation completed successfully, regardless of the return value of the non-blocking routine. When a non-blocking call is issued, certain data is saved in the MCAPI engine. This data is released only when MCAPI verifies you know the operation has completed. You can only determine this through the primitives described below.

MCAPI Primitives to Manage Non-Blocking Operations

The following MCAPI primitives are used to manage non-blocking operations:

- [mcapi_cancel](#)
- [mcapi_test](#)
- [mcapi_wait](#)
- [mcapi_wait_any](#)

Chapter 3

Configuring MGC MCAPI

This section describes the user level compile-time configuration options available within MGC MCAPI.

Topology Configuration

All compile-time topology configuration data structures are located in the *mcapi/mcapi_cfg.c* file. This file is used to configure interfaces on the local node and to set up routes to reachable nodes.

Interface Configuration

The global array `MCAPI_Int_Init_List` is populated with the initialization routine called to initialize each interface on the node. The final entry in the array must contain the value `MCAPI_NULL` to indicate the end of the data.

Route Configuration

The global array `MCAPI_Route_List` is populated with the Node ID of the target node and name of the interface used to reach the respective node. The final entry in the array must contain the tuple `<0, MCAPI_NULL>` to indicate the end of the data.

Compile-Time Macros

All user-configurable macros are located in the *mcapi/mcapi_cfg.h* file. The macros in this file are the only MCAPI macros you can modify.

`MCAPI_BUF_COUNT`

The total number of buffers to allocate for sending and receiving data. This value is used to allocate loopback buffers when loopback is enabled.

`MCAPI_DEFAULT_PRIO`

The default priority to use for all new endpoints created in the system.

MCAPI_ENABLE_FORWARD

Set this macro to `MCAPI_TRUE` to enable data forwarding on the local node.

When enabled, if a packet is received on the local node that is not destined to the node, the node forwards the packet to the intended destination.

When disabled, any data received by this node that is not intended for the node will be discarded.

MCAPI_ENABLE_LOOPBACK

Set this macro to `MCAPI_TRUE` to enable the internal loopback interface for sending and receiving data to and from the local node.

When enabled, the system allocates `MCAPI_BUF_COUNT` buffers of size `MCAPI_MAX_DATA_LEN` at the call to `mcapi_initialize` for sending and receiving local data.

MCAPI_ENDPOINT_PORT_INIT

The starting value of the global port counter used to assign a port ID to an endpoint when the endpoint is created with port ID `MCAPI_PORT_ANY`.

MCAPI_FREE_REQUEST_COUNT

The total number of free request structures available for processing incoming "get endpoint" requests from foreign nodes.

MCAPI_INT_NAME_LEN

The maximum length of an interface name.

MCAPI_INTERFACE_COUNT

The total number of interfaces on this node, including the loopback interface, if loopback is enabled.

MCAPI_MAX_DATA_LEN

The length of each loopback buffer created at initialization in the system. This value is used to allocate loopback buffers when loopback is enabled.

MCAPI_MAX_ENDPOINTS

The maximum number of simultaneously open endpoints allowed on this node.

MCAPI_PRIO_COUNT

The total number of priorities supported by this node. Set this value to the maximum count configured on the interface that supports the most priorities. For example, if interface 'x' supports 1 priority and interface 'y' supports 2 priorities, set this value to 2.

MCAPI_ROUTE_COUNT

The total number of routes on this node, including a route for the loopback interface, if loopback is enabled.

MCAPI_RX_CONTROL_PORT

The port ID of the local endpoint used to receive control messages from other nodes in the system. This value must match across all nodes in the system.

Chapter 4

Function Reference

This chapter describes the following types of functions:

- [Endpoint Primitives](#)
- [Message Primitives](#)
- [Node Primitives](#)
- [Packet Channel Primitives](#)
- [Primitives to Manage Non-Blocking Operations](#)
- [Scalar Channel Primitives](#)

Endpoint Primitives

This section provides a detailed reference of the following endpoint primitives:

- [mcapi_create_endpoint](#)
- [mcapi_delete_endpoint](#)
- [mcapi_get_endpoint](#)
- [mcapi_get_endpoint_attribute](#)
- [mcapi_set_endpoint_attribute](#)
- [mcapi_get_endpoint_i](#)

Related Topics

[Function Reference](#)

mcapi_create_endpoint

Include File: *mcapi_externs.h*

This function creates an endpoint on the local node for sending and/or receiving data. The return value represents a globally unique handle for referencing the endpoint.

Usage

```
mcapi_endpoint_t mcapi_create_endpoint( mcapi_port_t   port_id,  
                                       mcapi_status_t *mcapi_status)
```

Arguments

- **port_id**
The port ID of the endpoint. Each endpoint on a node must be assigned a unique port ID. When **port_id** is set to **MCAPI_PORT_ANY**, a port ID is assigned by MCAPI.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENDP_ISCREATED**
An endpoint with the specified **port_id** has already been created on this node.
- **MCAPI_ENODE_NOTINIT**
The local node has not been successfully initialized.
- **MCAPI_EENDP_LIMIT**
The maximum number of endpoints allowed in the system has been reached.
- **MCAPI_EEP_NOTALLOWED**
Endpoints cannot be created on this node, because **MCAPI_MAX_ENDPOINTS** is set to zero.
- **MCAPI_EPORT_NOTVALID**
MCAPI_PORT_ANY has been specified as the **port_id**, but there are no unused ports available on the node.

Example

```
mcapi_status_t   mcapi_status;  
mcapi_port_t     port_id;  
mcapi_endpoint_t endpoint;  
  
/* Create a new endpoint for sending and receiving messages. */  
endpoint = mcapi_create_endpoint(port_id, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)
```

```
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Endpoint Primitives](#)

mcapi_delete_endpoint

Include File: *mcapi_extrns.h*

This function deletes an endpoint. An endpoint can be deleted only by the node that created it.

Usage

```
void mcapi_delete_endpoint( mcapi_endpoint_t endpoint,  
                           mcapi_status_t    *mcapi_status)
```

Arguments

- **endpoint**
The target endpoint to delete.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The endpoint has not been created on the local node.
- **MCAPI_ECHAN_OPEN**
The endpoint is part of a connected channel and cannot be deleted.
- **MCAPI_ENOT_OWNER**
The caller is not the node that created the endpoint.

Description

All pending messages are discarded upon deletion. If the endpoint was part of an open channel connection, the connection must first be closed before deletion. If the endpoint is part of a connection that is only half open (if, for example, connect has been called but the send or receive side has not yet opened, or the send or receive side has opened but connect has not been called), the endpoint will be deleted successfully.

Example

```
mcapi_status_t    mcapi_status;  
mcapi_endpoint_t endpoint;  
  
/* Delete the endpoint. */  
mcapi_delete_endpoint(endpoint, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

```
}
```

Related Topics

[Endpoint Primitives](#)

mcapi_get_endpoint

Include File: *mcapi_extrns.h*

This function returns the endpoint handle associated with the specified node ID and port ID tuple.

Usage

```
mcapi_endpoint_t mcapi_get_endpoint( mcapi_node_t  node_id,  
                                     mcapi_port_t  port_id,  
                                     capi_status_t *mcapi_status)
```

Arguments

- **node_id**
The node ID on which the target endpoint was created.
- **port_id**
The port ID of the target endpoint.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENODE_NOT_VALID**
The *node_id* is invalid.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a blocking routine and suspends until the requested endpoint is created.

Note



This function can be used to retrieve the endpoint handle of an endpoint created on a foreign node. This functionality is useful for synchronizing nodes at boot up.

Example

```
mcapi_status_t  mcapi_status;  
mcapi_endpoint_t endpoint;  
  
/* Get the handle for an endpoint on a foreign node. */  
endpoint = mcapi_get_endpoint( MCAPI_Foreign_Node, MCAPI_Foreign_RX_Port,  
                              &mcapi_status);
```

```
if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Endpoint Primitives](#)

mcapi_get_endpoint_attribute

Include File: *mcapi_extrns.h*

This function retrieves an attribute associated with a local endpoint. Attributes can be retrieved for endpoints at any time.

Usage

```
void mcapi_get_endpoint_attribute( mcapi_endpoint_t endpoint,
                                  mcapi_uint_t      attribute_num,
                                  void               *attribute,
                                  size_t             attribute_size,
                                  mcapi_status_t     *mcapi_status)
```

Arguments

- **endpoint**
The target endpoint for which the attribute is being queried.
- **attribute_num**
The attribute to retrieve.

attribute_num	Description	attribute_size
MCAPI_ATTR_ENDP_PRIO	The priority of the endpoint.	mcapi_uint32_t
MCAPI_ATTR_NO_BUFFERS	The number of buffers allocated at initialization for sending and receiving data on the endpoint.	mcapi_uint32_t
MCAPI_ATTR_BUFFER_SIZE	The maximum buffer size for outgoing data.	mcapi_uint32_t
MCAPI_ATTR_RECV_BUFFERS_AVAILABLE	The number of buffers currently available for receiving data on this endpoint.	mcapi_uint32_t

- **attribute**
A pointer to memory that will be filled in with the value of the specific attribute upon successful completion of the call.
- **attribute_size**
The size of the input parameter *attribute.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The endpoint has not been created on the local node.
- **MCAPI_EATTR_NUM**
The attribute number is invalid.
- **MCAPI_EATTR_SIZE**
The attribute size is invalid.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t endpoint;
mcapi_uint32_t    priority;

/* Determine the priority associated with the endpoint. */
mcapi_get_endpoint_attribute(endpoint, MCAPI_ATTR_ENDP_PRIO,
                             (void*)&priority, sizeof(priority),
                             &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Endpoint Primitives](#)

mcapi_set_endpoint_attribute

Include File: *mcapi_extrns.h*

This function sets an attribute for a local endpoint. Attributes must be set before a channel connection is made.

Usage

```
void mcapi_set_endpoint_attribute(mcapi_endpoint_t endpoint,  
                                mcapi_uint_t      attribute_num,  
                                void               *attribute,  
                                size_t             attribute_size,  
                                mcapi_status_t     *mcapi_status)
```

Arguments

- **endpoint**
The target endpoint for which the attribute is being set.
- **attribute_num**
The attribute to set.

attribute_num	Description	attribute_size
MCAPI_ATTR_ENDP_PRIO	The priority of the endpoint.	mcapi_uint32_t

- **attribute**
A pointer to the new value of the attribute.
- **attribute_size**
The size of the input parameter *attribute.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The endpoint has not been created on the local node.
- **MCAPI_EATTR_NUM**
The attribute number is invalid.
- **MCAPI_EATTR_SIZE**
The attribute size is invalid.

- **MCAPI_ECONNECTED**
The endpoint is part of a connected channel, and attribute changes for connected endpoints are forbidden.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.
- **MCAPI_EREAD_ONLY**
The attribute is read-only.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  endpoint;
mcapi_uint32_t    priority = 0;

/* Set the priority associated with the endpoint. */
mcapi_set_endpoint_attribute(endpoint, MCAPI_ATTR_ENDP_PRIO,
                             (void*)&priority, sizeof(priority),
                             &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Endpoint Primitives](#)

mcapi_get_endpoint_i

Include File: *mcapi_externs.h*

This function retrieves the endpoint handle associated with the specified node ID and port ID tuple.

Usage

```
void mcapi_get_endpoint_i(mcapi_node_t    node_id,  
                          mcapi_port_t    port_id,  
                          mcapi_endpoint_t *endpoint,  
                          mcapi_request_t *request,  
                          mcapi_status_t  *mcapi_status)
```

Arguments

- **node_id**
The node ID on which the target endpoint was created.
- **port_id**
The port ID of the target endpoint.
- **endpoint**
A pointer to memory that will be filled in with the endpoint handle upon successful completion of the call.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENODE_NOT_VALID**
The `node_id` is invalid.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking routine and returns immediately.

The `*endpoint` and `*request` parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

Note



This function can be used to retrieve the endpoint handle of an endpoint created on a foreign node. This functionality is useful for synchronizing nodes at boot up.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  endpoint;
mcapi_request_t   request;
size_t           size;

/* Get the handle for an endpoint on a foreign node. */
mcapi_get_endpoint_i(MCAPI_Foreign_Node, MCAPI_Foreign_RX_Port,
                    &endpoint, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    . . .
}
```

Related Topics

[Endpoint Primitives](#)

Message Primitives

This section provides a detailed reference of the following messaging primitives:

- [mcapi_msg_available](#)
- [mcapi_msg_recv](#)
- [mcapi_msg_recv_i](#)
- [mcapi_msg_send](#)
- [mcapi_msg_send_i](#)

Related Topics

[Function Reference](#)

mcapi_msg_available

Include File: *mcapi_externs.h*

This function checks if a message is available on a local endpoint.

Usage

```
mcapi_uint_t mcapi_msg_available(mcapi_endpoint_t receive_endpoint,  
                                mcapi_status_t    *mcapi_status)
```

Arguments

- **receive_endpoint**
The endpoint for which data is being checked.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The receive endpoint has not been created on the local node.
- **MCAPI_ECONNECTED**
The receive endpoint is part of a connected channel.

Description

This is a non-blocking function and returns immediately.

Upon successful completion of the service, the number of available messages is returned; that is the number of receive calls that will complete successfully.

Example

```
mcapi_status_t    mcapi_status;  
mcapi_endpoint_t receive_endpoint;  
mcapi_uint_t      msg_count;  
  
/* Determine how many incoming messages are pending on the endpoint. */  
msg_count = mcapi_msg_available(receive_endpoint, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Message Primitives](#)

mcapi_msg_recv

Include File: *mcapi_extrns.h*

This function receives a connectionless message.

Usage

```
void mcapi_msg_recv(mcapi_endpoint_t receive_endpoint,  
                   void *buffer,  
                   size_t buffer_size,  
                   size_t *received_size,  
                   mcapi_status_t *mcapi_status)
```

Arguments

- **receive_endpoint**
The local endpoint on which data is being received.
- **buffer**
A pointer to the application buffer that will be filled in with the received data.
- **buffer_size**
The number of bytes that will fit in the application buffer.
- **received_size**
A pointer to the number of bytes that were copied into the application buffer.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The receive endpoint has not been created on the local node.
- **MCAPI_ETRUNCATED**
The incoming message size exceeds the application buffer size.
- **MCAPI_ECONNECTED**
The receive endpoint is part of a connected channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a blocking function and returns once the application buffer has been filled in with an incoming message. The application buffer must not be reused until the function returns.

There is currently no method in the MCAPI specification to learn the source endpoint of the received message.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  receive_endpoint;
char              buffer[128];
size_t            received_size;

/* Block for an incoming message. */
mcapi_msg_rcv(receive_endpoint, buffer, 128, &received_size,
               &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Message Primitives](#)

mcapi_msg_rcv_i

Include File: *mcapi_externs.h*

This function receives a connectionless message from any endpoint.

Usage

```
void mcapi_msg_rcv_i(mcapi_endpoint_t receive_endpoint,  
                    void *buffer,  
                    size_t buffer_size,  
                    mcapi_request_t *request,  
                    mcapi_status_t *mcapi_status)
```

Arguments

- **receive_endpoint**
The local endpoint on which data is being received.
- **buffer**
A pointer to the application buffer that will be filled in with the received data.
- **buffer_size**
The number of bytes that will fit in the application buffer.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The receive endpoint has not been created on the local node.
- **MCAPI_ETRUNCATED**
The incoming message size exceeds the application buffer size.
- **MCAPI_ECONNECTED**
The receive endpoint is part of a connected channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `*buffer` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

There is currently no method in the MCAPI specification to learn the source endpoint of the received message.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  receive_endpoint;
char              buffer[128];
mcapi_request_t   request;
size_t            size;

/* Issue the call to receive a message on the endpoint. */
mcapi_msg_rcv_i(receive_endpoint, buffer, 128, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for a message to be received. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Message Primitives](#)

mcapi_msg_send

Include File: *mcapi_extrns.h*

This function transmits a connectionless message to and from a specified set of endpoints. This is a blocking function and returns when the application buffer can be reused by the caller.

Usage

```
void mcapi_msg_send(mcapi_endpoint_t send_endpoint,  
                   mcapi_endpoint_t receive_endpoint,  
                   void *buffer,  
                   size_t buffer_size,  
                   mcapi_priority_t priority,  
                   mcapi_status_t *mcapi_status)
```

Arguments

- **send_endpoint**
The local endpoint from which data is being transmitted.
- **receive_endpoint**
The destination endpoint to which data is being transmitted.
- **buffer**
A pointer to the application buffer of data to transmit.
- **buffer_size**
The number of bytes being transmitted.
- **priority**
The priority of the outgoing buffer of data.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The send endpoint has not been created on the local node.
- **MCAPI_ENODE_NOT_VALID**
There is no route to the specified destination endpoint.
- **MCAPI_ECONNECTED**
The send endpoint is part of a connected channel.

- **MCAPI_EMESS_LIMIT**
The message size exceeds the maximum size allowed by the MCAPI implementation.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ENO_MEM**
There is inadequate memory available to complete the operation.
- **MCAPI_EPRIO**
The priority value is not supported.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  send_endpoint, receive_endpoint;
char buffer[128];

/* Send a message using the default priority level. */
mcapi_msg_send(send_endpoint, receive_endpoint, buffer, 128,
               MCAPI_DEFAULT_PRIO, &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Message Primitives](#)

mcapi_msg_send_i

Include File: *mcapi_extrns.h*

This function transmits a connectionless message to and from a specified set of endpoints.

Usage

```
void mcapi_msg_send_i(mcapi_endpoint_t send_endpoint,  
                      mcapi_endpoint_t receive_endpoint,  
                      void              *buffer,  
                      size_t            buffer_size,  
                      mcapi_priority_t priority,  
                      mcapi_request_t  *request,  
                      mcapi_status_t   *mcapi_status)
```

Arguments

- **send_endpoint**
The local endpoint from which data is being transmitted.
- **receive_endpoint**
The destination endpoint to which data is being transmitted.
- **buffer**
A pointer to the application buffer of data to transmit.
- **buffer_size**
The number of bytes being transmitted.
- **priority**
The priority of the outgoing buffer of data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
The send endpoint has not been created on the local node.
- **MCAPI_ENODE_NOT_VALID**
There is no route to the specified destination endpoint.

- **MCAPI_ECONNECTED**
The send endpoint is part of a connected channel.
- **MCAPI_EMESS_LIMIT**
The message size exceeds the maximum size allowed by the MCAPI implementation.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ENO_MEM**
There is inadequate memory available to complete the operation.
- **MCAPI_EPRIO**
The priority value is not supported.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and will return immediately.

The **buffer* and **request* input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns **MCAPI_INCOMPLETE**. A call to [mcapi_cancel](#) cancels the outstanding request.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  send_endpoint, receive_endpoint;
char              buffer[128];
mcapi_request_t   request;
size_t            size;

/* Send a message using the default priority level. */
mcapi_msg_send_i(send_endpoint, receive_endpoint, buffer, 128,
                  MCAPI_DEFAULT_PRIO, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the message to be transmitted. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Message Primitives](#)

Node Primitives

This section provides a detailed reference of the following endpoint primitives:

- [mcapi_initialize](#)
- [mcapi_finalize](#)
- [mcapi_get_node_id](#)

Related Topics

[Function Reference](#)

mcapi_initialize

Include File: *mcapi_extrns.h*

This function initializes all system resources required for an MCAPI node.

Usage

```
void mcapi_initialize(mcapi_node_t    node_id,  
                    mcapi_version_t *mcapi_version,  
                    mcapi_status_t  *mcapi_status)
```

Arguments

- **node_id**
The node ID of the node being initialized.
- **mcapi_version**
A pointer to memory that will be filled in with the version of the MCAPI software being used on this node.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_INITIALIZED**
The MCAPI environment has already been initialized for this node.
- **MCAPI_ENODE_NOT_VALID**
The value specified by *node_id* is in use by another node in the system.
- **MCAPI_EPARAM**
Invalid *mcapi_version* pointer.
- **MCAPI_ENO_MEM**
There is not enough system memory to complete initialization.

Description

Upon successful completion of this call, the node can use all MCAPI primitives for sending and receive data, configuring endpoints, and so on.

All interfaces and routes are initialized as per the compile-time configuration specified in *mcapi_cfg.c*.

It is an error to call this routine multiple times without first calling [mcapi_finalize](#).

Example

```
mcapi_status_t mcapi_status;  
mcapi_version_t mcapi_version;  
  
/* Initialize this node. */  
mcapi_initialize(MCAPI_LOCAL_NODE_ID, &mcapi_version, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[mcapi_finalize](#)

[Node Primitives](#)

mcapi_finalize

Include File: *mcapi_extrns.h*

This function shuts down MCAPI for the specific node and frees all system resources previously allocated by [mcapi_initialize](#).

Usage

```
void mcapi_finalize(mcapi_status_t *mcapi_status)
```

Arguments

- `mcapi_status`
A pointer to memory that will be filled in with the status of the call.

Return Values

- `MCAPI_SUCCESS`
The call was successful.
- `MCAPI_ENO_FINAL`
The MCAPI environment could not be shut down.

Description

All interfaces are stopped and corresponding resources freed, all MCAPI threads are deleted. Any open connections are gracefully closed and all outstanding buffers returned to the appropriate free list.

A subsequent call to [mcapi_initialize](#) will successfully initialize a shut down system.

Example

```
mcapi_status_t mcapi_status;  
  
/* Shut down MCAPI on this node. */  
mcapi_finalize(&mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[mcapi_initialize](#)

[Node Primitives](#)

mcapi_get_node_id

Include File: *mcapi_extrns.h*

This function returns the node ID registered for the local node by the call to [mcapi_initialize](#).

Usage

```
mcapi_uint_t mcapi_get_node_id(mcapi_status_t *mcapi_status)
```

Arguments

- `mcapi_status`

A pointer to memory that will be filled in with the status of the call.

Return Values

- `MCAPI_SUCCESS`
The call was successful.
- `MCAPI_ENODE_NOTINIT`
The local node has not been successfully initialized.

Example

```
mcapi_status_t mcapi_status;  
mcapi_uint_t   node_id;  
  
/* Get the node ID of the local node. */  
node_id = mcapi_get_node_id(&mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Node Primitives](#)

Packet Channel Primitives

This section provides a detailed reference of the following packet channel primitives:

- [mcapi_connect_pktchan_i](#)
- [mcapi_open_pktchan_recv_i](#)
- [mcapi_open_pktchan_send_i](#)
- [mcapi_packetchan_recv_close_i](#)
- [mcapi_packetchan_send_close_i](#)
- [mcapi_pktchan_available](#)
- [mcapi_pktchan_free](#)
- [mcapi_pktchan_recv](#)
- [mcapi_pktchan_recv_i](#)
- [mcapi_pktchan_send](#)
- [mcapi_pktchan_send_i](#)

Related Topics

[Function Reference](#)

mcapi_connect_pktchan_i

Include File: *mcapi_extrns.h*

This function connects two endpoints over a unidirectional packet channel in which one endpoint is designated as the sender and the other as the receiver.

Usage

```
void mcapi_connect_pktchan_i(mcapi_endpoint_t send_endpoint,  
                             mcapi_endpoint_t receive_endpoint,  
                             mcapi_request_t *request,  
                             mcapi_status_t *mcapi_status)
```

Arguments

- **send_endpoint**
The endpoint in the connection that will be transmitting data.
- **receive_endpoint**
The endpoint in the connection that will be receiving data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of **MCAPI_SUCCESS** still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
One of the endpoints has not been created on the respective node.
- **MCAPI_ECONNECTED**
One of the endpoints has already been connected over a channel.
- **MCAPI_EATTR_INCOMP**
One or more attributes of the two endpoints to be connected does not match.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) cancels the outstanding request.

The connect operation can be issued by the sender, receiver, or by a third party node. Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, and the receive endpoint must issue a call to open the receive side of the connection.

The connect call and respective open calls can be issued in any order.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  send_endpoint, receive_endpoint;
mcapi_request_t   request;
size_t            size;

/* Connect two endpoints over a packet channel. */
mcapi_connect_pktchan_i(send_endpoint, receive_endpoint, &request,
                        &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connect call to complete. */
    mcapi\_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_open_pktchan_rcv_i

Include File: *mcapi_externs.h*

This function opens the receive side of a packet channel.

Usage

```
void mcapi_open_pktchan_rcv_i(  
    mcapi_pktchan_rcv_hdl_t *rcv_handle,  
    mcapi_endpoint_t         receive_endpoint,  
    mcapi_request_t          *request,  
    mcapi_status_t           *mcapi_status)
```

Arguments

- **rcv_handle**
A pointer to the receive handle that will be filled in upon successful completion of the call.
- **receive_endpoint**
The local endpoint in the connection that will be receiving data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful. Data can now be sent and received across the channel.
- **MCAPI_ENOT_ENDP**
The receive endpoint has not been created on the local node.
- **MCAPI_ENOT_CONNECTED**
A connect call has not been issued for the connection yet. This is not a failure, but an informational message to inform the user of the connection status.
- **MCAPI_ECHAN_TYPE**
The receive endpoint has already been connected over a scalar channel.
- **MCAPI_EDIR**
The receive endpoint has been connected as a send endpoint.

- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `*rcv_handle` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, the receive endpoint must issue a call to open the receive side of the connection and a connect call must be issued to connect the two endpoints.

Although the connect call and respective open calls can be issued in any order, the open calls will not report success until the connect call has been issued.

Example

```
mcapi_status_t          mcapi_status;
mcapi_endpoint_t        receive_endpoint;
mcapi_request_t         request;
size_t                  size;
mcapi_pktchan_rcv_hdl_t rcv_handle;

/* Open the receive side of the packet channel. */
mcapi_open_pktchan_rcv_i(&rcv_handle, receive_endpoint, &request,
                        &mcapi_status);

if ( (mcapi_status == MCAPI_SUCCESS) ||
     (mcapi_status == MCAPI_ENOT_CONNECTED) )
{
    /* Wait for the connection to be fully opened. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_open_pktchan_send_i

Include File: *mcapi_extrns.h*

This function opens the send side of a packet channel.

Usage

```
void mcapi_open_pktchan_send_i(mcapi_pktchan_send_hdl_t *send_handle,
                               mcapi_endpoint_t          send_endpoint,
                               mcapi_request_t            *request,
                               mcapi_status_t             *mcapi_status)
```

Arguments

- **send_handle**
A pointer to the send handle that will be filled in upon successful completion of the call.
- **send_endpoint**
The local endpoint in the connection that will be sending data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful. Data can now be sent and received across the channel.
- **MCAPI_ENOT_ENDP**
The send endpoint has not been created on the local node.
- **MCAPI_ENOT_CONNECTED**
A connect call has not been issued for the connection yet. This is not a failure, but an informational message to inform the user of the connection status.
- **MCAPI_ECHAN_TYPE**
The send endpoint has already been connected over a scalar channel.
- **MCAPI_EDIR**
The send endpoint has been connected as a receive endpoint.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `*send_handle` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, the receive endpoint must issue a call to open the receive side of the connection and a connect call must be issued to connect the two endpoints. Although the connect call and respective open calls can be issued in any order, the open calls will not report success until the connect call has been issued.

Example

```
mcapi_status_t          mcapi_status;
mcapi_endpoint_t        send_endpoint;
mcapi_request_t         request;
size_t                  size;
mcapi_pktchan_send_hdl_t send_handle;

/* Open the send side of the packet channel. */
mcapi_open_pktchan_send_i(&send_handle, send_endpoint, &request,
                          &mcapi_status);

if ( (mcapi_status == MCAPI_SUCCESS) ||
      (mcapi_status == MCAPI_ENOT_CONNECTED) )
{
    /* Wait for the connection to be fully opened. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_packetchan_recv_close_i

Include File: *mcapi_extrns.h*

This function closes the receive side of a packet channel.

Usage

```
void mcapi_packetchan_recv_close_i(
    mcapi_pktchan_recv_hdl_t receive_handle,
    mcapi_request_t           *request,
    mcapi_status_t            *mcapi_status)
```

Arguments

- **receive_handle**
The local receive handle for which the receive side is being closed.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The receive handle does not reference a valid packet channel handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ENOT_OPEN**
The packet connection has not been opened fully; that is, the connect call has not been made to connect the two endpoints.
- **MCAPI_ECHAN_TYPE**
The handle is for a scalar channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately. The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or

[mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) will cancel the outstanding request.

This call can be made only by the receive side of a connection.

Example

```
mcapi_status_t      mcapi_status;
mcapi_request_t     request;
size_t             size;
mcapi_pktchan_rcv_hdl_t rcv_handle;

/* Close the receive side of the packet channel. */
mcapi_packetchan_rcv_close_i(rcv_handle, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connection to be fully closed. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_packetchan_send_close_i

Include File: *mcapi_extrns.h*

This function closes the send side of a packet channel.

Usage

```
void mcapi_packetchan_send_close_i(
    mcapi_pktchan_send_hndl_t send_handle,
    mcapi_request_t             *request,
    mcapi_status_t              *mcapi_status)
```

Arguments

- **send_handle**
The local send handle for which the send side is being closed.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The send handle does not reference a valid packet channel handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENOT_OPEN**
The packet connection has not been opened fully; that is, the connect call has not been made to connect the two endpoints.
- **MCAPI_ECHAN_TYPE**
The handle is for a scalar channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) cancels the outstanding request.

This call can be made only by the send side of a connection.

Example

```
mcapi_status_t      mcapi_status;
mcapi_request_t     request;
size_t             size;
mcapi_pktchan_send_hndl_t send_handle;

/* Close the send side of the packet channel. */
mcapi_pktchan_send_close_i(send_handle, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connection to be fully closed. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_available

Include File: *mcapi_extrns.h*

This function checks if a packet is available on a local connected packet receive handle.

Usage

```
mcapi_uint_t mcapi_pktchan_available(  
                                mcapi_pktchan_rcv_hdl_t receive_handle,  
                                mcapi_status_t          *mcapi_status)
```

Arguments

- **receive_handle**
The local packet channel receive handle for which data is being checked.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The receive handle does not reference a valid handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ENOT_CONNECTED**
The connection is not fully open yet.
- **MCAPI_ECHAN_TYPE**
The receive handle is part of a scalar connection.

Description

This is a non-blocking function and returns immediately.

Upon successful completion of the service, the number of available messages is returned; that is, the number of receive calls that will complete successfully.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_pktchan_rcv_hdl_t receive_handle;  
mcapi_uint_t           msg_count;  
  
/* Determine how many incoming packets are pending on the connection. */  
msg_count = mcapi_pktchan_available(receive_handle, &mcapi_status);
```

```
if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_free

Include File: *mcapi_extrns.h*

This function frees a system buffer that was received over a packet channel connection.

Usage

```
void mcapi_pktchan_free(void          *buffer,  
                        mcapi_status_t *mcapi_status)
```

Arguments

- **buffer**
A pointer to the system buffer to return to the available buffer list.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_ENOT_VALID_BUF**
The buffer is not a valid system buffer.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

Buffers can be freed in any order. Failure to free all received buffers will result in irreversible system buffer loss. Buffers pending receipt on an endpoint are freed automatically upon closing the receive side of the connection.

Example

```
mcapi_status_t      mcapi_status;  
char                *buffer;  
size_t              received_size;  
mcapi_pktchan_rcv_hdl_t rcv_handle;  
  
/* Receive a packet across the connected channel. */  
mcapi_pktchan_rcv(rcv_handle, (void**)&buffer, &received_size,  
                  &mcapi_status);  
  
if (mcapi_status == MCAPI_SUCCESS)  
{  
    /* Process the data. */  
    . . .  
  
    /* Free the buffer. */  
    mcapi_pktchan_free(buffer, &mcapi_status);  
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_rcv

Include File: *mcapi_extrns.h*

This function receives a packet across a connected packet channel.

Usage

```
void mcapi_pktchan_rcv(mcapi_pktchan_rcv_hdl_t receive_handle,  
                      void **buffer,  
                      size_t *received_size,  
                      mcapi_status_t *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **buffer**
A pointer to a pointer that will be set to the application payload region of an incoming driver buffer.
- **received_size**
A pointer to memory that will be filled in with the number of bytes received on the connection.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ECHAN_TYPE**
The receive handle is part of a scalar connection.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a blocking function and returns when the application buffer pointer has been set to point at an incoming packet.

The ****buffer** and ***received_size** input parameters must not be modified or reused by the application until the call returns.

Example

```
mcapi_status_t      mcapi_status;
char                *buffer;
size_t              received_size;
mcapi_pktchan_recv_hndl_t recv_handle;

/* Receive a packet across the connected channel. */
mcapi_pktchan_recv(recv_handle, (void**)&buffer, &received_size,
                   &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_rcv_i

Include File: *mcapi_extrns.h*

This function receives a packet across a connected packet channel.

Usage

```
void mcapi_pktchan_rcv_i(mcapi_pktchan_rcv_hdl_t receive_handle,  
                        void **buffer,  
                        mcapi_request_t *request,  
                        mcapi_status_t *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **buffer**
A pointer to a pointer that will be set to the application payload region of an incoming driver buffer.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of **MCAPI_SUCCESS** still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the receive operation and the number of bytes received.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ECHAN_TYPE**
The receive handle is part of a scalar connection.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `**buffer` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

This operation provides zero copy functionality by passing the incoming driver buffer directly to the application. The application is responsible for freeing the buffer via a call to [mcapi_pktchan_free](#) when finished processing the incoming data.

Failure to correctly free incoming buffers results in irreversible system buffer loss.

Example

```
mcapi_status_t      mcapi_status;
char                *buffer;
mcapi_request_t     request;
size_t              size;
mcapi_pktchan_rcv_hndl_t rcv_handle;

/* Receive a packet across the connected channel. */
mcapi_pktchan_rcv_i(rcv_handle, (void**)&buffer, &request,
                    &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for a packet to be received. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_send

Include File: *mcapi_extrns.h*

This function transmits a packet across a connected packet channel. This is a blocking function and returns when the application buffer can be reused by the caller.

Usage

```
void mcapi_pktchan_send(mcapi_pktchan_send_hndl_t send_handle,  
                        void *buffer,  
                        size_t size,  
                        mcapi_status_t *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **buffer**
A pointer to the application buffer of data to transmit.
- **size**
The number of bytes being transmitted.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ECHAN_TYPE**
The send handle is part of a scalar connection.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_EPACK_LIMIT**
The packet size exceeds the maximum size allowed by the MCAPI implementation.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ENO_MEM**
There is inadequate memory available to complete the operation.

- **MCAPI_EPARAM**

One of the pointer input arguments is NULL.

Example

```
mcapi_status_t          mcapi_status;
mcapi_pktchan_send_hdl_t send_handle;
char                    buffer[128];

/* Send a packet across the connected channel. */
mcapi_pktchan_send(send_handle, buffer, 128, &mcapi_status);

if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

mcapi_pktchan_send_i

Include File: *mcapi_extrns.h*

This function transmits a packet across a connected packet channel.

Usage

```
void mcapi_pktchan_send_i(mcapi_pktchan_send_hndl_t send_handle,  
                           void *buffer,  
                           size_t size,  
                           mcapi_request_t *request,  
                           mcapi_status_t *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **buffer**
A pointer to the application buffer of data to transmit.
- **size**
The number of bytes being transmitted.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ECHAN_TYPE**
The send handle is part of a scalar connection.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_EPACK_LIMIT**
The packet size exceeds the maximum size allowed by the MCAPI implementation.

- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ENO_MEM**
There is inadequate memory available to complete the operation.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and will return immediately.

The **buffer* and **request* input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns **MCAPI_INCOMPLETE**. A call to [mcapi_cancel](#) cancels the outstanding request.

Example

```
mcapi_status_t          mcapi_status;
mcapi_pktchan_send_hdl_t send_handle;
char                    buffer[128];
mcapi_request_t         request;
size_t                  size;

/* Send a packet across the connected channel. */
mcapi_pktchan_send_i(send_handle, buffer, 128, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the packet to be transmitted. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Packet Channel Primitives](#)

Primitives to Manage Non-Blocking Operations

This section provides a detailed reference of the following non-blocking operations:

- [mcapi_cancel](#)
- [mcapi_test](#)
- [mcapi_wait](#)
- [mcapi_wait_any](#)

Related Topics

[Function Reference](#)

mcapi_cancel

Include File: *mcapi_extrns.h*

This function cancels a pending non-blocking operation.

Usage

```
void mcapi_cancel(mcapi_request_t *request,  
                 mcapi_status_t *mcapi_status)
```

Arguments

- **request**
A pointer to the pending non-blocking operation to cancel.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOTREQ_HANDLE**
The request is invalid.

Example

```
mcapi_status_t    mcapi_status;  
mcapi_endpoint_t  receive_endpoint;  
char              buffer[128];  
mcapi_request_t   request;  
  
/* Issue a non-blocking receive call. */  
mcapi_msg_rcv_i(receive_endpoint, buffer, 128, &request, &mcapi_status);  
  
if (mcapi_status == MCAPI_SUCCESS)  
{  
    /* Cancel the receive operation. */  
    mcapi_cancel(&request, &mcapi_status);  
  
    if (mcapi_status != MCAPI_SUCCESS)  
    {  
        /* Report the error. */  
        . . .  
    }  
}
```

Related Topics

[Primitives to Manage Non-Blocking Operations](#)

mcapi_test

Include File: *mcapi_extrns.h*

This function tests if a non-blocking operation has completed. This function is non-blocking and returns immediately.

Usage

```
mcapi_boolean_t mcapi_test(mcapi_request_t *request,
                           size_t          *size,
                           mcapi_status_t  *mcapi_status)
```

Arguments

- **request**
A pointer to the request structure associated with the pending non-blocking operation being tested for completion.
- **size**
If the operation being tested for completion is a send or receive operation, this memory will be filled in with the number of bytes sent or received by the respective operation.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_TRUE**
The operation has completed successfully.
- **MCAPI_ENOTREQ_HANDLE**
The request input parameter is invalid.
- **MCAPI_INCOMPLETE**
The operation is still pending completion.
- **MCAPI_EREQ_CANCELED**
The operation has been canceled.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.
- **MCAPI_FALSE**
The operation did not complete successfully.

Example

```
mcapi_status_t  mcapi_status;
mcapi_endpoint_t receive_endpoint;
char            buffer[128];
mcapi_request_t request;
```

```
size_t      size;
mcapi_boolean_t finished;

/* Issue a non-blocking receive call. */
mcapi_msg_rcv_i(receive_endpoint, buffer, 128, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Check if data has been received. */
    finished = mcapi_test(&request, &size, &mcapi_status);

    if ( (finished == MCAPI_TRUE) && (mcapi_status == MCAPI_SUCCESS) )
    {
        /* Process the data. */
        . . .
    }
}
```

Related Topics

[Primitives to Manage Non-Blocking Operations](#)

mcapi_wait

Include File: *mcapi_extrns.h*

This function waits for a non-blocking operation to complete.

Usage

```
mcapi_boolean_t mcapi_wait(mcapi_request_t *request,  
                           size_t          *size,  
                           mcapi_status_t  *mcapi_status,  
                           mcapi_timeout_t timeout)
```

Arguments

- **request**
A pointer to the request structure associated with the pending non-blocking operation being tested for completion.
- **size**
If the operation being tested for completion is a send or receive operation, this memory will be filled in with the number of bytes sent or received by the respective operation.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.
- **timeout**
The number of milliseconds to wait for the operation to complete.

Return Values

- **MCAPI_TRUE**
The operation has completed successfully.
- **MCAPI_ENOTREQ_HANDLE**
The request input parameter is invalid.
- **MCAPI_EREQ_CANCELED**
The operation has been canceled.
- **MCAPI_EREQ_TIMEOUT**
The timeout occurred before the operation completed.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.
- **MCAPI_FALSE**
The operation did not complete successfully.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  receive_endpoint;
char              buffer[128];
mcapi_request_t   request;
size_t            size;
mcapi_boolean_t   finished;

/* Issue a non-blocking receive call. */
mcapi_msg_rcv_i(receive_endpoint, buffer, 128, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for data to be received. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);

    if ( (finished == MCAPI_TRUE) && (mcapi_status == MCAPI_SUCCESS) )
    {
        /* Process the data. */
        . . .
    }
}
```

Related Topics

[Primitives to Manage Non-Blocking Operations](#)

mcapi_wait_any

Include File: *mcapi_extrns.h*

This function waits for any non-blocking operation in a list to complete. If one of the operations completes (either successfully or due to an error), the function returns the index into the list of requests of the operation that completed.

Usage

```
mcapi_int_t mcapi_wait_any(size_t      number,  
                           mcapi_request_t **requests,  
                           size_t      *size,  
                           mcapi_status_t *mcapi_status,  
                           mcapi_timeout_t timeout)
```

Arguments

- **number**
The number of operations in the requests array.
- **requests**
A pointer to an array of request structures associated with the pending non-blocking operations being tested for completion.
- **size**
If the index of the operation returned to the caller is a send or receive operation, this memory will be filled in with the number of bytes sent or received by the respective operation.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.
- **timeout**
The number of milliseconds to wait for any one of the operations to complete.

Return Values

- **MCAPI_SUCCESS**
An operation has completed successfully.
- **MCAPI_ENOTREQ_HANDLE**
One or more of the requests in the request array is invalid.
- **MCAPI_EREQ_CANCELED**
The operation referenced by the return index has been canceled.
- **MCAPI_EREQ_TIMEOUT**
The timeout occurred before any of the operations completed.

- **MCAPI_EPARAM**

One of the pointer input arguments is NULL.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  receive_endpoint1, receive_endpoint2;
char              buffer1[128], buffer2[128];
mcapi_request_t   request1, request2;
mcapi_request_t   *requests[2];
size_t           size;
mcapi_int_t       index;

/* Issue a non-blocking receive call on the first endpoint. */
mcapi_msg_rcv_i(receive_endpoint1, buffer1, 128, &request1,
                &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Issue a non-blocking receive call on the second endpoint. */
    mcapi_msg_rcv_i(receive_endpoint2, buffer2, 128, &request2,
                    &mcapi_status);

    /* Populate the request array. */
    requests[0] = &request1;
    requests[1] = &request2;

    /* Wait for data to be received on either endpoint. */
    mcapi_wait_any(2, requests, &size, &mcapi_status, MCAPI_INFINITE);

    if (mcapi_status == MCAPI_SUCCESS)
    {
        /* Process the data. */
        . . .
    }
}
```

Related Topics

[Primitives to Manage Non-Blocking Operations](#)

Scalar Channel Primitives

This section provides a detailed reference of the following scalar channel primitives:

- [mcapi_connect_sclchan_i](#)
- [mcapi_open_sclchan_recv_i](#)
- [mcapi_open_sclchan_send_i](#)
- [mcapi_sclchan_available](#)
- [mcapi_sclchan_recv_close_i](#)
- [mcapi_sclchan_recv_uint8](#)
- [mcapi_sclchan_recv_uint16](#)
- [mcapi_sclchan_recv_uint32](#)
- [mcapi_sclchan_recv_uint64](#)
- [mcapi_sclchan_send_close_i](#)
- [mcapi_sclchan_send_uint8](#)
- [mcapi_sclchan_send_uint16](#)
- [mcapi_sclchan_send_uint32](#)
- [mcapi_sclchan_send_uint64](#)

Related Topics

[Function Reference](#)

mcapi_connect_sclchan_i

Include File: *mcapi_extrns.h*

This function connects two endpoints over a unidirectional scalar channel in which one endpoint is designated as the sender and the other as the receiver.

Usage

```
void mcapi_connect_sclchan_i(mcapi_endpoint_t send_endpoint,  
                             mcapi_endpoint_t receive_endpoint,  
                             mcapi_request_t *request,  
                             mcapi_status_t *mcapi_status)
```

Arguments

- **send_endpoint**
The endpoint in the connection that will be transmitting data.
- **receive_endpoint**
The endpoint in the connection that will be receiving data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_ENDP**
One of the endpoints has not been created on the respective node.
- **MCAPI_ECONNECTED**
One of the endpoints has already been connected over a channel.
- **MCAPI_EATTR_INCOMP**
One or more attributes of the two endpoints to be connected does not match.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) cancels the outstanding request.

The connect operation can be issued by the sender, receiver, or by a third party node. Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, and the receive endpoint must issue a call to open the receive side of the connection. The connect call and respective open calls can be issued in any order.

Example

```
mcapi_status_t    mcapi_status;
mcapi_endpoint_t  send_endpoint, receive_endpoint;
mcapi_request_t   request;
size_t            size;

/* Connect two endpoints over a scalar channel. */
mcapi_connect_sclchan_i(send_endpoint, receive_endpoint, &request,
                        &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connect call to complete. */
    mcapi\_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_open_sclchan_recv_i

Include File: *mcapi_extrns.h*

This function opens the receive side of a scalar channel.

Usage

```
void mcapi_open_sclchan_recv_i(  
                                mcapi_sclchan_recv_hndl_t *receive_handle,  
                                mcapi_endpoint_t           receive_endpoint,  
                                mcapi_request_t            *request,  
                                mcapi_status_t             *mcapi_status)
```

Arguments

- **receive_handle**
A pointer to the receive handle that will be filled in upon successful completion of the call.
- **receive_endpoint**
The local endpoint in the connection that will be receiving data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful. Data can now be sent and received across the channel.
- **MCAPI_ENOT_ENDP**
The receive endpoint has not been created on the local node.
- **MCAPI_ENOT_CONNECTED**
A connect call has not been issued for the connection yet. This is not a failure, but an informational message to inform the user of the connection status.
- **MCAPI_ECHAN_TYPE**
The receive endpoint has already been connected over a packet channel.
- **MCAPI_EDIR**
The receive endpoint has been connected as a send endpoint.

- **MCAPI_EPARAM**

One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `*receive_handle` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, the receive endpoint must issue a call to open the receive side of the connection and a connect call must be issued to connect the two endpoints. Although the connect call and respective open calls can be issued in any order, the open calls will not report success until the connect call has been issued.

Example

```
mcapi_status_t          mcapi_status;
mcapi_endpoint_t        receive_endpoint;
mcapi_request_t         request;
size_t                  size;
mcapi_sclchan_rcv_hndl_t rcv_handle;

/* Open the receive side of the scalar channel. */
mcapi_open_sclchan_rcv_i(&rcv_handle, receive_endpoint, &request,
                        &mcapi_status);

if ( (mcapi_status == MCAPI_SUCCESS) ||
     (mcapi_status == MCAPI_ENOT_CONNECTED) )
{
    /* Wait for the connection to be fully opened. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_open_sclchan_send_i

Include File: *mcapi_externs.h*

This function opens the send side of a scalar channel.

Usage

```
void mcapi_open_sclchan_send_i(mcapi_sclchan_send_hndl_t *send_handle,  
                               mcapi_endpoint_t           send_endpoint,  
                               mcapi_request_t            *request,  
                               mcapi_status_t            *mcapi_status)
```

Arguments

- **send_handle**
A pointer to the send handle that will be filled in upon successful completion of the call.
- **send_endpoint**
The local endpoint in the connection that will be sending data.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call. A status of `MCAPI_SUCCESS` still requires a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#).

Return Values

- **MCAPI_SUCCESS**
The call was successful. Data can now be sent and received across the channel.
- **MCAPI_ENOT_ENDP**
The send endpoint has not been created on the local node.
- **MCAPI_ENOT_CONNECTED**
A connect call has not been issued for the connection yet. This is not a failure, but an informational message to inform the user of the connection status.
- **MCAPI_ECHAN_TYPE**
The send endpoint has already been connected over a packet channel.
- **MCAPI_EDIR**
The send endpoint has been connected as a receive endpoint.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The `*send_handle` and `*request` input parameters must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns `MCAPI_INCOMPLETE`. A call to [mcapi_cancel](#) cancels the outstanding request.

Before data can be exchanged across the connection, the send endpoint must issue a call to open the send side of the connection, the receive endpoint must issue a call to open the receive side of the connection and a connect call must be issued to connect the two endpoints. Although the connect call and respective open calls can be issued in any order, the open calls will not report success until the connect call has been issued.

Example

```
mcapi_status_t          mcapi_status;
mcapi_endpoint_t        end_endpoint;
mcapi_request_t         request;
size_t                  size;
mcapi_sclchan_send_hdl_t send_handle;

/* Open the send side of the scalar channel. */
mcapi_open_sclchan_send_i(&send_handle, send_endpoint, &request,
                          &mcapi_status);

if ( (mcapi_status == MCAPI_SUCCESS) ||
      (mcapi_status == MCAPI_ENOT_CONNECTED) )
{
    /* Wait for the connection to be fully opened. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}

else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_available

Include File: *mcapi_extrns.h*

This function checks if data is available on a local connected scalar receive handle.

Usage

```
mcapi_uint_t mcapi_sclchan_available(  
                                mcapi_sclchan_rcv_hdl_t receive_handle,  
                                mcapi_status_t          *mcapi_status)
```

Arguments

- **receive_handle**
The local scalar channel receive handle for which data is being checked.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The receive handle does not reference a valid handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ENOT_CONNECTED**
The connection is not fully open yet.
- **MCAPI_ECHAN_TYPE**
The receive handle is part of a packet connection.

Description

This is a non-blocking function and returns immediately. Upon successful completion of the service, the number of available scalars is returned; that is, the number of receive calls that will complete successfully.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_sclchan_rcv_hdl_t receive_handle;  
mcapi_uint_t           msg_count;  
  
/* Determine how many incoming scalars are pending on the connection. */  
msg_count = mcapi_sclchan_available(receive_handle, &mcapi_status);
```

```
if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_rcv_close_i

Include File: *mcapi_externs.h*

This function closes the receive side of a scalar channel.

Usage

```
void mcapi_sclchan_rcv_close_i(mcapi_sclchan_rcv_hndl_t receive_handle,  
                               mcapi_request_t          *request,  
                               mcapi_status_t           *mcapi_status)
```

Arguments

- **receive_handle**
The local receive handle for which the receive side is being closed.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The receive handle does not reference a valid packet scalar handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ENOT_OPEN**
The scalar connection has not been opened fully; that is, the connect call has not been made to connect the two endpoints.
- **MCAPI_ECHAN_TYPE**
The handle is for a packet channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) cancels the outstanding request.

This call can be made only by the receive side of a connection.

Example

```
mcapi_status_t      mcapi_status;
mcapi_request_t     request;
size_t             size;
mcapi_sclchan_rcv_hndl_t rcv_handle;

/* Close the receive side of the scalar channel. */
mcapi_sclchan_rcv_close_i(rcv_handle, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connection to be fully closed. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}
else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_recv_uint8

Include File: *mcapi_extrns.h*

This function receives a 8-bit scalar across a connected scalar channel. This is a blocking function and returns when data has been successfully received.

Usage

```
mcapi_uint8_t mcapi_sclchan_recv_uint8(  
                                mcapi_sclchan_recv_hndl_t receive_handle,  
                                mcapi_status_t *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ESCL_SIZE**
The scalar available on the connection is not 8-bits in size.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t      mcapi_status;  
mcapi_sclchan_recv_hndl_t receive_handle;  
mcapi_uint8_t      scalar;  
  
/* Receive a 8-bit scalar on the connected channel. */  
scalar = mcapi_sclchan_recv_uint8(receive_handle, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_rcv_uint16

Include File: *mcapi_extrns.h*

This function receives a 16-bit scalar across a connected scalar channel. This is a blocking function and returns when data has been successfully received.

Usage

```
mcapi_uint16_t mcapi_sclchan_rcv_uint16(  
                                mcapi_sclchan_rcv_hndl_t receive_handle,  
                                mcapi_status_t           *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ESCL_SIZE**
The scalar available on the connection is not 16-bits in size.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t           mcapi_status;  
mcapi_sclchan_rcv_hndl_t receive_handle;  
mcapi_uint16_t           scalar;  
  
/* Receive a 16-bit scalar on the connected channel. */  
scalar = mcapi_sclchan_rcv_uint16(receive_handle, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```


Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_rcv_uint32

Include File: *mcapi_externs.h*

This function receives a 32-bit scalar across a connected scalar channel. This is a blocking function and returns when data has been successfully received.

Usage

```
mcapi_uint32_t mcapi_sclchan_rcv_uint32(  
                                mcapi_sclchan_rcv_hndl_t receive_handle,  
                                mcapi_status_t *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ESCL_SIZE**
The scalar available on the connection is not 32-bits in size.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t      mcapi_status;  
mcapi_sclchan_rcv_hndl_t receive_handle;  
mcapi_uint32_t      scalar;  
  
/* Receive a 32-bit scalar on the connected channel. */  
scalar = mcapi_sclchan_rcv_uint32(receive_handle, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_rcv_uint64

Include File: *mcapi_externs.h*

This function receives a 64-bit scalar across a connected scalar channel. This is a blocking function and returns when data has been successfully received.

Usage

```
mcapi_uint64_t mcapi_sclchan_rcv_uint64(  
                                mcapi_sclchan_rcv_hndl_t receive_handle,  
                                mcapi_status_t *mcapi_status)
```

Arguments

- **receive_handle**
The local handle on which data is being received.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The receive handle provided is a send handle.
- **MCAPI_ESCL_SIZE**
The scalar available on the connection is not 64-bits in size.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t      mcapi_status;  
mcapi_sclchan_rcv_hndl_t receive_handle;  
mcapi_uint64_t      scalar;  
  
/* Receive a 64-bit scalar on the connected channel. */  
scalar = mcapi_sclchan_rcv_uint64(receive_handle, &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
    . . .  
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_send_close_i

Include File: *mcapi_extrns.h*

This function closes the send side of a scalar channel.

Usage

```
void mcapi_sclchan_send_close_i(mcapi_sclchan_send_hdl_t send_handle,  
                                mcapi_request_t           *request,  
                                mcapi_status_t            *mcapi_status)
```

Arguments

- **send_handle**
The local send handle for which the send side is being closed.
- **request**
A pointer to the request structure to be passed into [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) to determine the status of the call. This opaque data structure must not be read or written to by the application.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The send handle does not reference a valid scalar channel handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENOT_OPEN**
The scalar connection has not been opened fully; that is, the connect call has not been made to connect the two endpoints.
- **MCAPI_ECHAN_TYPE**
The handle is for a packet channel.
- **MCAPI_EPARAM**
One of the pointer input arguments is NULL.

Description

This is a non-blocking function and returns immediately.

The **request* input parameter must not be modified or reused by the application until a subsequent call to [mcapi_test](#), [mcapi_wait](#), or [mcapi_wait_any](#) no longer returns MCAPI_INCOMPLETE. A call to [mcapi_cancel](#) cancels the outstanding request.

This call can be made only by the send side of a connection.

Example

```
mcapi_status_t      mcapi_status;
mcapi_request_t     request;
size_t             size;
mcapi_sclchan_send_hndl_t send_handle;

/* Close the send side of the scalar channel. */
mcapi_sclchan_send_close_i(send_handle, &request, &mcapi_status);

if (mcapi_status == MCAPI_SUCCESS)
{
    /* Wait for the connection to be fully closed. */
    mcapi_wait(&request, &size, &mcapi_status, MCAPI_INFINITE);
}
else
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_send_uint8

Include File: *mcapi_extrns.h*

This function transmits a 8-bit scalar across a connected scalar channel. This is a blocking function and returns when the data has been successfully transmitted.

Usage

```
void mcapi_sclchan_send_uint8(mcapi_sclchan_send_hndl_t send_handle,  
                             mcapi_uint8_t          dataword,  
                             mcapi_status_t          *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **dataword**
The 8-bit scalar being transmitted.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_sclchan_send_hndl_t send_handle;  
mcapi_uint8_t          scalar;  
  
/* Send a 8-bit scalar across the connected channel. */  
mcapi_sclchan_send_uint8(send_handle, scalar &mcapi_status);
```



```
if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_send_uint16

Include File: *mcapi_extrns.h*

This function transmits a 16-bit scalar across a connected scalar channel. This is a blocking function and returns when the data has been successfully transmitted.

Usage

```
void mcapi_sclchan_send_uint16(mcapi_sclchan_send_hndl_t send_handle,  
                               mcapi_uint16_t          dataword,  
                               mcapi_status_t          *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **dataword**
The 16-bit scalar being transmitted.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_sclchan_send_hndl_t send_handle;  
mcapi_uint16_t          scalar;  
  
/* Send a 16-bit scalar across the connected channel. */  
mcapi_sclchan_send_uint16(send_handle, scalar &mcapi_status);
```

```
if (mcapi_status != MCAPI_SUCCESS)
{
    /* Report the error. */
    . . .
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_send_uint32

Include File: *mcapi_extrns.h*

This function transmits a 32-bit scalar across a connected scalar channel. This is a blocking function and returns when the data has been successfully transmitted.

Usage

```
void mcapi_sclchan_send_uint32(mcapi_sclchan_send_hdl_t send_handle,  
                               mcapi_uint32_t          dataword,  
                               mcapi_status_t          *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **dataword**
The 32-bit scalar being transmitted.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_sclchan_send_hdl_t send_handle;  
mcapi_uint32_t          scalar;  
  
/* Send a 32-bit scalar across the connected channel. */  
mcapi_sclchan_send_uint32(send_handle, scalar &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
}
```

```
    . . .  
}
```

Related Topics

[Scalar Channel Primitives](#)

mcapi_sclchan_send_uint64

Include File: *mcapi_extrns.h*

This function transmits a 64-bit scalar across a connected scalar channel. This is a blocking function and returns when the data has been successfully transmitted.

Usage

```
void mcapi_sclchan_send_uint64(mcapi_sclchan_send_hdl_t send_handle,  
                               mcapi_uint64_t          dataword,  
                               mcapi_status_t          *mcapi_status)
```

Arguments

- **send_handle**
The local handle from which data is being transmitted.
- **dataword**
The 64-bit scalar being transmitted.
- **mcapi_status**
A pointer to memory that will be filled in with the status of the call.

Return Values

- **MCAPI_SUCCESS**
The call was successful.
- **MCAPI_ENOT_HANDLE**
The handle does not reference a valid scalar connection handle.
- **MCAPI_EDIR**
The send handle provided is a receive handle.
- **MCAPI_ENO_BUFFER**
There are no available buffers for transmitting data.
- **MCAPI_ECHAN_TYPE**
The handle has been connected as a packet channel.

Example

```
mcapi_status_t          mcapi_status;  
mcapi_sclchan_send_hdl_t send_handle;  
mcapi_uint64_t          scalar;  
  
/* Send a 64-bit scalar across the connected channel. */  
mcapi_sclchan_send_uint64(send_handle, scalar &mcapi_status);  
  
if (mcapi_status != MCAPI_SUCCESS)  
{  
    /* Report the error. */  
}
```

```
    . . .  
}
```

Related Topics

[Scalar Channel Primitives](#)

Chapter 5

Porting MGC MCAPI

MGC MCAPI currently offers porting layers for using MCAPI with the Linux and Nucleus PLUS operating systems. This section describes the porting efforts required to support another target operating system.

OS-Specific Directory Structure

All OS-specific files are stored in an OS-labeled directory at `/mcapi/porting` directory. Each OS specific directory contains two files named according to the OS; *mcapi_os_XYZ.c* and *mcapi_os_XYZ.h*.

Note



The Nucleus header file is located in */include*.

The source file stores the porting layer functions called by MCAPI to perform various operating system functions. The header file holds definitions for MCAPI data structures.

To begin the porting work, copy one of the existing OS-specific directories and rename the directory and files according to the new operating system.

Porting the Data Structures

Most of the MCAPI data structures defined in the OS-specific header file are straightforward to port. This section describes those data structures that warrant further commentary.

MCAPI_POINTER

This data structure is important when using a shared memory driver for data transmission. The list of global buffer structures is stored in shared memory and accessed by multiple possibly architecturally different cores. It is important that each CPU read the data properly; therefore, MCAPI_POINTER is used to denote the size of the pointers in the MCAPI_BUFFER structure.

Set this to the largest value supported across cores. For example, if core 0 uses 32-bit pointers and core 1 uses 64-bit pointers, both cores must set MCAPI_POINTER to `mcapi_uint64_t`; otherwise, the data will not be properly accessed by core 0.

mcapi_cond_t

This data structure is used to suspend and resume threads within MCAPI as described further down in this section.

Initialization and Shut Down

The following routines are used to initialize and shut down the MCAPI node.

MCAPI_Init_OS

This routine is called from [mcapi_initialize](#) when a node is initialized. Upon return from this call, the control task used to process control messages from foreign nodes, [mcapi_process_ctrl_msg\(\)](#), must be up and running.

MCAPI_Cleanup_Task

This routine is called from the control task when the application has issued a call to [mcapi_shutdown\(\)](#). If the OS supports the ability for a task to clean up after itself, this routine must be modified to perform clean up functionality.

MCAPI_Exit_OS

This routine is called from [mcapi_finalize](#) when a node is shut down. Upon return from this call, all resources that were allocated by [MCAPI_Init_OS](#) must be shut down and cleaned up.

Protecting Data

MCAPI uses one node-local structure to protect the node-local data structures from simultaneous access. While the routines are named '_Mutex,' an implementation is free to use any sort of data structure to accomplish this functionality.

MCAPI_Create_Mutex

This routine is called from [mcapi_initialize](#) to create the MCAPI node-local structure used for protecting MCAPI data.

MCAPI_Delete_Mutex

This routine is called from [mcapi_finalize](#) to destroy the MCAPI node-local structure used for protecting MCAPI data.

MCAPI_Obtain_Mutex

This routine is called within MCAPI to acquire the MCAPI node-local structure used for protecting MCAPI data.

MCAPI_Release_Mutex

This routine is called within MCAPI to release the MCAPI node-local structure used for protecting MCAPI data.

Delivering Data to MCAPI

It is important to understand how data is delivered to MCAPI from the underlying interface driver when making MCAPI porting decisions. When a driver receives an incoming buffer, the final goal is to deliver the data to the appropriate application thread. This involves parsing the incoming data to determine the destination endpoint, finding the appropriate endpoint structure, and resuming any application threads suspended on the operation.

When deciding how data will make its way from the driver to the application, consider the constraints of the underlying interface driver. [Figure 5-1](#) shows, for example, that in Nucleus, data is received in the context of a HISR. Because a HISR cannot block or perform many other OS operations, the Nucleus shared memory driver cannot identify the appropriate endpoint and wake the respective threads. The HISR is only able to set an event to trigger some other thread to perform this work. Therefore, the MCAPI porting layer for Nucleus PLUS creates a thread at initialization for processing incoming data, `mcapi_process_input()`, which waits for an event to be set by the interface driver. When data is received, the driver calls [MCAPI_Set_RX_Event](#) to set the event, then `mcapi_process_input()` wakes and invokes `mcapi_rx_data()`, the MCAPI generic routine that handles delivering data to the application.

Figure 5-1. Nucleus Data Delivery and MCAPI

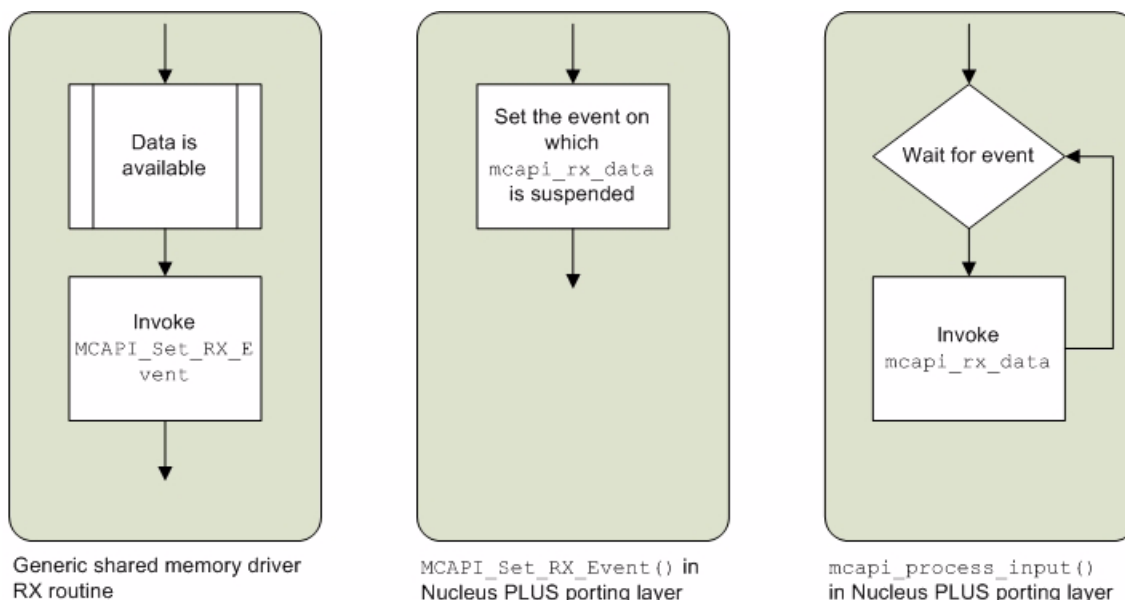
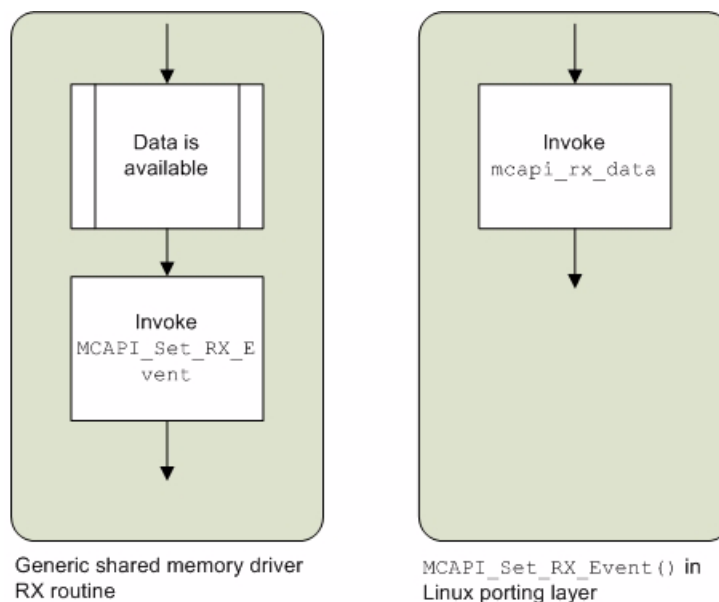


Figure 5-2 shows the model for Linux is different from Nucleus. Within Linux, the shared memory driver receives data from the kernel driver within the context of a thread; therefore the shared memory driver is able to perform all intermediate processing. When the driver calls `MCAPI_Set_RX_Event`, `mcapi_rx_data()` is invoked directly.

Figure 5-2. Linux Data Delivery and MCAPI



MCAPI_Set_RX_Event

This routine is called from the interface driver to indicate that data is ready to be received on the node. If the interface driver is able to process data directly from its thread of execution, this routine invokes `mcapi_rx_data()` as described above. Otherwise, a separate thread must be created by [MCAPI_Init_OS](#) for processing data, and [MCAPI_Set_RX_Event](#) must resume that thread which will then invoke `mcapi_rx_data()`.

MCAPI_Lock_RX_Queue

This routine is called from `mcapi_rx_data()` to obtain protection for the global data shared between MCAPI and the interface driver. If data is received by the interface driver within a HISR, this routine must lock out interrupts; otherwise, the routine can just obtain the global MCAPI mutex.

MCAPI_Unlock_RX_Queue

This routine is called from `mcapi_rx_data()` to release protection for the global data shared between MCAPI and the interface driver. If data is received by the interface driver within a HISR, this routine must restore the previous interrupt level; otherwise, the routine can just release the global MCAPI mutex.

Suspending / Resuming Threads

The previous section describes the flow of incoming data from the interface driver to the application layer. When trying to receive data at the application layer, a thread may suspend pending incoming data. When data is received, the thread must be resumed in some way.

Most operating systems offer several mechanisms for suspending and resuming a thread; that is, events, conditions, mutexes, semaphores, etc. You determine the most effective suspension method for the new OS.

The MCAPI porting layer uses the data structure `MCAPI_COND_STRUCT` for suspending and resuming threads:

```
typedef struct
{
    mcapi_cond_t mcapi_cond;
    mcapi_cond_t *mcapi_cond_ptr;
} MCAPI_COND_STRUCT;
```

MCAPI_Init_Condition

This routine is called from [mcapi_wait_any](#) to initialize the `mcapi_cond` parameter of the `MCAPI_COND_STRUCT` data structure. The `mcapi_cond` parameter is later used to resume the suspended thread.

MCAPI_Set_Condition

This routine is called from [mcapi_wait_any](#) to enable multiple threads to suspend on the same `mcapi_cond` parameter. This routine uses the `mcapi_cond_t*` parameter of the `MCAPI_COND_STRUCT` data structure as follows:

```
request->mcapi_cond.mcapi_cond_ptr = &condition->mcapi_cond;
```

Note



Unless the target OS implements suspension much different than the methods described here, this function requires no modification.

MCAPI_Clear_Condition

This routine is called from [mcapi_wait_any](#) to clear the `mcapi_cond_t*` parameter of the `MCAPI_COND_STRUCT` data structure as follows:

```
request->mcapi_cond.mcapi_cond_ptr = MCAPI_NULL;
```

Note



Unless the target OS implements suspension much different than the methods described here, this function requires no modification.

MCAPI_Suspend_Task

This routine is called from MCAPI to suspend an application thread pending completion or cancellation of some operation. When called from [mcapi_wait_any](#), no request structure is passed into the routine. Otherwise, a request structure is passed into the routine, and the routine must first initialize the `mcapi_cond` of the request structure and add the request structure to the node-local queue `node_data->mcapi_local_req_queue` before suspending.

The routine must support a method of timing out the suspension given a millisecond timeout.

Upon completion, the routine must destroy the `condition->mcapi_cond` parameter.

MCAPI_Resume_Task

This routine is called from `mcapi_resume()` to resume an application thread that has suspended pending some operation. If the `mcapi_cond_ptr` parameter of the request structure is not `MCAPI_NULL`, the routine operates on `mcapi_cond_ptr`. Otherwise, the routine operates on `mcapi_cond` to resume the suspended thread.

Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:
 - 1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;
 - 1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;
 - 1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;
 - 1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;
 - 1.5. "Embedded Software" means Software that is embeddable;
 - 1.6. "End-User" means Customer's customer;
 - 1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;
 - 1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;
 - 1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;
 - 1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;
 - 1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;
 - 1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;
 - 1.13. "Products" means Software, Term-Licensed Products and/or Hardware;
 - 1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

- 1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;
- 1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;
- 1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;
- 1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;
- 1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;
- 1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;
- 1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;
- 1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and
- 1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. Orders, Fees and Payment.

- 2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. Grant of License.

- 3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**
- 3.2. License Type. The license type shall be identified in the applicable Addenda.
- 3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

- 3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.
- 3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.
- 3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. Beta Code.

- 4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. Restrictions on Use.

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.
- 5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.
- 5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. Support Services.

- 6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at <http://supportnet.mentor.com/about/legal/>.
- 6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at <http://www.mentor.com/codebench-support-legal>.
- 6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at <http://www.mentor.com/vsipl-support-legal>.
- 6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased

solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at <http://www.mentor.com/mel-support-legal>.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.
8. **Limited Warranty.**
 - 8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.
 - 8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR \$500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
10. **Hazardous Applications.**
 - 10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").
 - 10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.
 - 10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.
 - 10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. Infringement.

- 11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.
 - 11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.
 - 11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.
12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.
- 12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.
 - 12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.
13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.
14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.