Multicore Communications API (MCAPI) Specification V2.015

Document ID: MCAPI API Specification
Document Version: 2.015
Status: Release
Distribution: General

The Multicore Association, Inc.
PO Box 4854
El Dorado Hills, CA 95762
530-672-9113
www.multicore-association.org

Table of Contents

Preface

This document is intended to assist software developers who are either implementing multicore communication functions using MCAPI or writing applications that use MCAPI.

Definitions

AMP: Asymmetric multiprocessing, in which two or more processing cores having the same or different architecture may be running the same or different operating systems (or no OS at all).

API: Application programming interface.

Blocking: A blocking function does not return until the function has completed or resulted in an error. A blocking API call returns when the buffer may be reused by the application. A thread-suspension mechanism is required for blocking calls.

Channel: A unidirectional, point-to-point, FIFO connection between a pair of endpoints.

Connection-Based Communication: A method, used in packet and scalar streams, that potentially removes the message header and route-discovery overhead from each packet or scalar.

Domain: An implementation of MCAPI includes one or more domains, each with one or more nodes. The concept of domains is used consistently for all Multicore Associations APIs. A domain is comparable to a subnet in a network.

Endpoint: A destination to which messages may be sent, or to which a communication connection may be established. It is like a network socket. Each endpoint has a unique identifier that consists of the tuple {<domain ID>, <node ID>, <port ID>}. A node may have multiple endpoints.

Handle: An abstract reference by one node to an object managed by another node. Unlike a pointer, a handle does not contain a literal address.

IPC: Inter-processor communication.

MCA: The Multicore Association.

MCAPI: Multicore Communications API Specification, defined by The Multicore Association.

Message: A connectionless datagram sent from one endpoint to another. It is similar to a UDP datagram in networking.

MRAPI: Multicore Resource API Specification, defined by The Multicore Association.

MTAPI: Multicore Task API Specification, defined by The Multicore Association.

Node: An independent thread of control. It could be a process, thread, instance of an operating system, hardware accelerator, processor core, or other entity with an independent program counter. Each node can belong to only one domain. The concept of nodes applies consistently to all Multicore Associations APIs.

Non-Blocking: A non-blocking function returns in a timely manner (without having to block on any IPC to any remote nodes), but the requested transaction completes in a non-blocking manner.

Packet Stream: A FIFO sequence of data packets of variable size, sent from one endpoint to another after a connection has been established.

POSIX: Portable Operating System Interface, an API for Unix specified by the IEEE.

Scalar Stream: A FIFO sequence of single data words sent from one endpoint to another after a connection has been established. Each word may contain 8, 16, 32, or 64 bits, but the send and receive scalar size must be the same.

SMP: Symmetric multiprocessing, in which two or more identical processing cores are connected to a shared main memory and are controlled by a single OS instance.

SoC: System-on-chip.

Timely: An operation is timely if it returns without having to block on any IPC to any remote nodes.

Related Documents

- *Multicore Resource API (MRAPI) Specification*, The Multicore Association.
- *Multicore Task API (MTAPI) Specification*, The Multicore Association (in progress).
- *Multicore Programming Practices (MPP)*, The Multicore Association (in progress)..

1. Introduction

1.1 Overview

This Multicore Communications API (MCAPI) specification defines an API and a semantic for communication and synchronization between processing cores in embedded systems. It does not define which link management, device model, or wire protocol is used underneath it.

The multiple cores may be homogeneous or heterogeneous and located on a single chip or on multiple chips in a circuit board. MCAPI is scalable and can support virtually any number of cores, each with a different processing architecture and each running the same or a different operating system, or no OS at all. As such, MCAPI is intended to provide source-code compatibility that allows applications to be ported from one operating environment to another well into the future.

In the most basic MCAPI model, each core is represented as an individual “node” in the system. When a node needs to communicate with another node, it specifies an endpoint for sending or receiving data.

MCAPI defines three fundamental communications types:

- Messages: Connection-less datagrams, similar to UDP datagrams in networking.
- Packet Channels: Connection-oriented, unidirectional, FIFO packet streams.
- Scalar channel: Connection-oriented, single-word, unidirectional, FIFO scalar streams.

Each of these communications types have their own API calls. Messages are the most flexible form of communication, and they are useful when senders, receivers, and per-message priorities are dynamically changing. These are commonly used for synchronization, initialization, and load-balancing.

Packet and scalar channels provide light-weight, socket-like stream communication mechanisms for senders and receivers with static communication graphs. In a multicore application, MCAPI’s channel API functions provide an extremely low-overhead, ASIC-like, unidirectional FIFO communications capability. Channels are typically set up once at initialization, during which the MCAPI runtime system attempts to perform most of the work involved in communications between a specific pair of endpoints (such as name lookup, route determination, and buffer allocation). Subsequent sends and receives on the channel then incur the minimal overhead of physically transferring data. Packet channels support streaming communication of multiword data buffers. Scalar channels are optimized for sequences of scalar values. Channel API calls are simple and statically typed, thereby minimizing dynamic software overhead and allowing applications to access the underlying multicore hardware with extremely low latency and energy cost.

MCAPI’s objective is to provide a limited number of calls with sufficient communication functionality while keeping it simple enough to allow efficient implementations. Additional functionality can be layered on top of the API set. The calls are exemplifying functionality and are not mapped to any particular existing implementation.

1.2 History

Although multiprocessor designs have been around for decades, semiconductor vendors have only recently turned to multicore processors as a solution for the so-called Moore's Gap[1] effect. Modern multicore CPU design enables CPU performance to track Moore's law with attractive CPU power-consumption properties. This trend is causing many system designers, who previously uses single-core designs, to change to multicore designs in order to meet their performance and/or power-consumption requirements. This has a huge impact on the software architecture for those systems, which must now consider inter-processor communication (IPC) issues to pass data between the cores.

The MCAPI specification traces its heritage to communication APIs such as Message Passing Interface (MPI) and sockets. Both MPI and sockets were targeted primarily toward inter-computer communication, whereas MCAPI targs inter-core communication in a multicore chip. Accordingly, a principal design goal of MCAPI was to serve as a low-latency interface, leveraging efficient on-chip interconnect. MCAPI is thus a light-weight API whose communication latencies and memory footprint are expected to be significantly lower than that of MPI or sockets. However, because of the more limited scope of multicore communications and its goal of low latency, MCAPI is less flexible than MPI or sockets.

MCAPI can be distinguished from the Portable Operating System Interface (POSIX) threads, *pthreads*. MCAPI is a communications API for messaging and streaming, whereas pthreads is an API for parallelizing or threading applications. Pthreads offers a programming model that relies on shared memory for communication and locks for synchronization. In multicore processors with caches, efficient pthreads implementations require support for cache coherence. Because they lack modularity, lock-based parallel programs are hard to compose together. MCAPI, in which parallelism is specified using standard means such as processes or threads, offers an alternative and complementary parallel programming approach that is modular and composable. MCAPI allows simpler embedded multicore implementations because it works in multicore processors with private memory or with shared memory.

Inter-Processor Communication (IPC) for device software using multicore processors has some new requirements beyond those imposed on multiprocessor designs that use discrete processors. Within a multicore chip, the shorter distance for electrical signals enables data to be passed much more quickly, energy-efficiently, and reliably than between discrete processors on a board or over a backplane. In addition, the bandwidth available on chip is orders of magnitude greater. The low latency and high bandwidth offer the potential of ASIC-like high-speed communication capabilities between cores. This means that the focus of IPC for multicore processors must prioritize performance in terms of both latency and throughput. Also, some multicore processors will have many cores that rely on chip-internal memory or cache to execute code, to avoid the von Neumann bottleneck which multicore processors can suffer due to the multiple cores contending for external memory accesses. Cores that execute from chip-internal memory require an IPC implementation that has a small memory footprint. For these reasons, the two primary goals for a multicore IPC API design, such as MCAPI, are that the implementation of it can achieve extremely high performance and low memory footprint.

In order for a multicore IPC implementation to achieve high performance and tiny footprint, these two goals need to be immutable when in conflict with other desirable IPC API attributes, such as flexible physical or logical connectivity topology, ease of use, OS integration, or compatibility with existing IPC programming models or APIs.

MCAPI endeavors to meet these goals. Before MCAPI, no IPC API has been designed with these two immutable goals targeting multicore processors. MCAPI was designed to be sufficiently complete for many multicore application programming tasks and to serve as a solid foundation for sophisticated multicore software services. MCAPI can be used to implement distributed services such as name services and distributed work queues.

[1] Moore's Law has hit a wall where increased transistors per sequential CPU chip is no longer resulting in a linear increase in performance of that CPU chip.

2. MCAPI Concepts

The major MCAPI concepts are described in this section. For definitions of terms, see the Preface.

2.1 Domains

An MCAPI domain is comprised of one or more MCAPI nodes in a multicore topology, and it is used for routing purposes. The scope of a domain is implementation-defined; for example, its scope could be a single chip with multiple cores or multiple processor chips on a board. The domain ID is specified once at node initialization. Potential uses for domains include topologies that may change dynamically, include non-MCAPI sub-topologies, require separation between different transports, or have open and secure areas.

2.2 Nodes

An MCAPI node is an independent thread of control, such as a process, thread, processor, hardware accelerator, or instance of an operating system. A given MCAPI implementation specifies what kind of thing constitutes a node for that implementation.

The intent is not to have a mixture of node definitions in the same implementation (or in different domains within an implementation). Note that if a node is defined as a thread of execution with its private address space (like a process), a core with a single unprotected address space OS is equivalent to a node, whereas a core with a virtual memory OS can host multiple nodes.

The MCAPI standard aims to be implementable on both process-oriented and thread-oriented systems. MCAPI defines a transport layer between nodes. The MCAPI standard explicitly avoids having any dependence on a particular shared memory model or process-protection model. The communication operations defined by MCAPI should be implementable with identical semantics on either thread- or process- based systems. Thus, programs that use only the MCAPI APIs should be portable between MCAPI implementations. On the other hand, programs that make use of APIs outside of MCAPI (for example pthreads) will only be portable between systems that include those extra APIs.

A node ID is specified in the call to `mcapi_initialize()`.

The concept of nodes in MCAPI is shared with other Multicore Association API specifications. Therefore, implementations that support multiple MCA APIs must define a node in exactly the same way, and initialization of nodes across these APIs must be consistent. In the future, the Multicore Association will consider defining a small set of unified API calls and header files that enforce these semantics.

Things to Remember:

- The MCAPI domain and node numbering plan is established when the MCAPI communication topology is configured at design-time, so programmers need not worry about this at run-time.
- It is up to the MCAPI implementation to configure communication topology interfaces that enable communication between nodes in the communication topology.

2.3 Endpoints

MCAPI endpoints are socket-like communication-termination points. They are created with the `mcapi_endpoint_create()` function. A node can have multiple endpoints. Thus, an endpoint is a topology-global unique identifier. Endpoints are identified by a `<domain_id, node_id, port_id>` tuple. An endpoint is created by implicitly referring to the `node_ID` of the node on which the endpoint is being created, and explicitly specifying a `port_ID`. Alternatively, MCAPI allows the creation of an endpoint by requesting the next available endpoint. Static names allow the programmer to define a network topology at compile time and to embed the topology in the source code in a straightforward fashion. It also enables the use of external tools to generate topologies automatically and it facilitates simple initialization.

It is up to the implementation to ensure that the resulting endpoint reference is unique in the system. MCAPI allows a reference to endpoints to be obtained by a node other than the node whose `node_ID` is associated with the endpoint, using the `mcapi_endpoint_get()` function. This endpoint reference can be used as a destination for messages or to specify the opposite end of a connection. Endpoint references created by requesting the next available endpoint must be passed by the creating node to other nodes to facilitate communication. Only the creating node is allowed to receive from an endpoint.

Endpoints have a set of attributes. These attributes may be related to Quality of Service (QoS), buffers, timeouts, etc. Currently MCAPI defines a basic set of attribute numbers and their associated structure or data type, found in the `mcapi.h` file. The first 64 endpoint attributes (numbers and data types) are reserved for the Multicore Association (MCA). Vendors desiring to add vendor-specific endpoint attributes in their implementations can request, from the MCA, to be assigned ranges of 32 vendor-specific endpoint attribute numbers. MCAPI defines functions to get and set the attributes of endpoints. Attributes can only be set on local node endpoints but be gotten from local and remote node endpoints. It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (for now, whether attributes are compatible or not is implementation-defined). It is also an error to attempt to change the attributes of endpoints that are connected.

2.4 Channels

Channels provide point-to-point FIFO connections between a pair of endpoints. The channels are unidirectional. There are two types: packet channels and scalar channels. There is no separate channel object. Instead, channels can be referenced through the endpoints to which they are bound. Since channels are point-to-point connections between a pair of endpoints, either endpoint can be used to refer to the channel.

2.4.1 Communications

Applications in an MCAPI topology communicate with one another by sending data between communication endpoints. Three modes of communication are supported: connectionless messages, connection-oriented packet channels, or connection-oriented scalar channels. Each communication action in all three methods requires the specification of a pair of endpoints—a send endpoint and a receive endpoint—explicitly for messages and implicitly for channels using a handle. Messages and packet channels provide both blocking and non-blocking send and receive functions. The runtime system implements each of these logical communication actions over some physical medium that is supported by MCAPI (such as on-chip interconnect, bus, shared memory, or Ethernet) by automatically establishing a “link” to enable communication with the other node in the network. The runtime system also takes care of routing traffic over the appropriate link.

Messages and packet channels communicate using data buffers, whereas scalar channels transfer scalar values such as four-byte words.

From an application's perspective, a data buffer to be communicated is a byte string from 0 up to some maximum length determined by the amount of memory in the system. The internal structure of the data buffer is determined by the application. The communication byte order is not defined in this specification because MCAPI is an API specification. Since MCAPI does not currently define a wire protocol (there may be a future Multicore Association specification which does specify that) then endian-ness and alignment issues are not relevant to this specification.

Connectionless communication allows an endpoint to send or receive messages with one or more endpoints elsewhere in the communication topology. A given message can be sent to a single endpoint (unicast). Multicast and broadcast messaging modes are not supported by MCAPI, but can be built on top of it.

Connection-oriented communication allows an endpoint to establish a socket-like streaming connection to a peer endpoint elsewhere in the communication topology, and then send data to that peer. A connection is established using an explicit handshake mechanism prior to sending or receiving any application data. After a connection has been established, it remains active for highly efficient data transfers until it is terminated by one of the sides, or until the communication path between the endpoints is severed (for example, by the failure of the node or link which one of the endpoints is utilizing).

Connection-oriented communication over MCAPI channels is designed to be reliable, in that an application can send data over a connection and assume that the data will be delivered to the specified destination as long as that destination is reachable.

Things To Remember:

- In an MCAPI communication topology where different nodes may be running on different CPU types and/or operating systems, applications must ensure that the internal structure of a message is well-defined, and the applications must account for any differences in message content endian-ness, field size, and field alignment.

2.5 Data Delivery

On the surface, whether the communications use messages, packet channels, or scalar channels, the data delivery in MCAPI is a simple series of steps:

- A sender creates and sends the data (either a buffer for messages and packet channels, or a scalar value for scalar channels).
- The MCAPI implementation carries the data to the specified destination.
- The receiver receives and then consumes the data.

In practice, this is exactly what happens most of the time. However, there are several places along the way where things can get complicated, and in these cases it is important for application designers to understand exactly what MCAPI will do. The sections that follow describe the various steps performed by MCAPI during the transfer of a data packet.

2.5.1 Data Sending

The first step in sending data is to create it. The user application then sends the data using one of several mechanisms. The application supplies the data in a user buffer for messages and packet channels. The application supplies the data directly as a scalar variable for scalar channels.

The most common reason MCAPI is unable to send a piece of data is because the sender passes in one or more invalid arguments to the send routine. The term "invalid" refers both to values that are never acceptable under any circumstances (such as specifying a data buffer size exceeding the maximum size allowed for a specific implementation) and to values that are not acceptable for the current sender (such as an invalid endpoint handle, or an unconnected channel).

In all of these cases the send operation will return a failure code indicating that the intended data was not sent. If the data is sent successfully, the send operation returns a success indication. Success means that the entire buffer has been sent.

Things to Remember:

- If the sender specifies a destination address for a connectionless message, but the destination address does not currently exist within the MCAPI communication topology, MCAPI does NOT treat this as an invalid send request (i.e., it is not the sender's fault that the destination does not exist). Implementations may, however, treat this as an error condition, and if so the Implementation must document the specifics. Instead, MCAPI creates the message and then sends it. The return value for the send operation will indicate success since the message was successfully created and processed by MCAPI. MCAPI cannot return an error since the connectivity failure could be occurring elsewhere in the MCAPI communication topology.

2.5.2 Data Routing

After a send of some data has been requested, MCAPI determines what node the data should be sent to. The endpoint address indicates the destination node to which the packet should be sent. The data is then either handed off to the destination endpoint directly, if it is on the same node as the sender, or the data is passed to a link for off-node transmission to the destination node.

One problem that can arise during the routing phase of packet delivery is that no working link to the specified destination node can be found. In this case, the packet is discarded. Implementations may optionally report this is an error condition.

2.5.3 Data Consumption

When the data reaches the destination endpoint, it is added to an endpoint receive queue. The data typically remains in the endpoint's receive queue until it is received by the application that owns the endpoint. Queued data items are consumed by the application in a FIFO manner for channels, and in a FIFO order per priority level for messages.

For messages, the user application must supply a buffer into which the MCAPI runtime system fills in the data. For packet channels, on the other hand, the MCAPI runtime system supplies the buffer containing the data to the user. For messages, the application can use its data buffer after it is filled by MCAPI in any way it chooses. Specifically, the application can re-supply the data buffer to MCAPI to receive the next message. Data buffers supplied by MCAPI during packet channel receives can also be used by the application in any way it chooses. MCAPI reuses a data buffer it has supplied only after the user application has specifically freed the buffer using the `mcapi_pktchan_release()` function.

If an application terminates access to the before all data in the receive queue is consumed, all unconsumed data items are considered undeliverable and are discarded.

It is very important that MCAPI applications be engineered to consume their incoming data items at a rate that prevents them from accumulating in large numbers in any endpoint receive queue.

A FIFO model is used for the packet and scalar channel interfaces. FIFOs have limited storage, and when the storage is used up, an implementation can block until more storage is available or the implementation can return an error code.

Non-blocking send operations will still return in a timely fashion, but data transfer will not occur if the FIFO is full. The sender will block upon calling service routines to check for completion of a non-blocking operation, and will continue to block until a receive operation is performed or a timeout occurs.

The mcapi_endpoint_get/set attribute() APIs can be used to query or to control the amount of receive buffering that is provided for a given endpoint. Implementations may define a static receive-buffer size according to the amount of buffering provided in drivers and/or hardware. Alternatively, an implementation can be allowed to dynamically allocate storage until the system runs out of memory.

2.5.4 Waiting for Non-Blocking Operations

The connectionless-message and packet-channel API functions have both blocking and non-blocking variants. The non-blocking variants have "_i" appended to the function name to indicate that the function will return *immediately* and will complete in a non-blocking manner.

The non-blocking versions fill in an mcapi_request_t object and return control to the user before the communication operation is completed. The programmer can then use the mcapi_test(), mcapi_wait(), and mcapi_wait_any() functions to query the status of the non-blocking operation. These functions are non-destructive, meaning that no message, packet, or scalar is removed from the endpoint queue by the functions. The mcapi_test() function is non-blocking, whereas the mcapi_wait() and mcapi_wait_any() functions will block until the requested operation completes or a timeout occurs. Multiple threads should not be waiting for the same request; attempting to do so results in an error.

If a buffer of data is passed to a non-blocking operation (for example, to mcapi_msg_send_i() mcapi_msg_recv_i(), or to mcapi_pktchan_send_i()), that buffer should not be accessed by the user application for the duration of the non-blocking operation. That is, after a buffer has been passed to a non-blocking operation, the program may not read or write the buffer until mcapi_test(), mcapi_wait(), or mcapi_wait_any() have indicated completion, or until mcapi_cancel() has canceled the operation.

The MCAPI scalar channels API provides only blocking send and receive methods. Scalar channels are intended to provide a very low-overhead interface for moving a stream of values. Non-blocking operations add overhead. The sort of streaming algorithms that take advantage of scalar channels should not require a non-blocking send or receive method; each process should simply receive a value to work on, do its work, send the result out on a channel, and repeat. Applications that require non-blocking semantics should use packet channels instead of scalar channels.

2.6 Messages

MCAPI messages, shown in Figure 1, provide a flexible method to transmit data between endpoints without first establishing a connection. The buffers on both sender and receiver sides must be provided by the user application. MCAPI messages may be sent with different priorities, on a per-message basis. It is not allowed to send a message to a connected endpoint. Implementations may prevent messages from being sent to a connected endpoint or to leave it up to the application to manage this. Functionality for this may be added in a future version of MCAPI, and it is therefore recommended that implementations preventing messages from being sent to a connected endpoint use the MCAPI_ERR_GENERAL error code to report an error. The implementation-specific behavior should be documented.

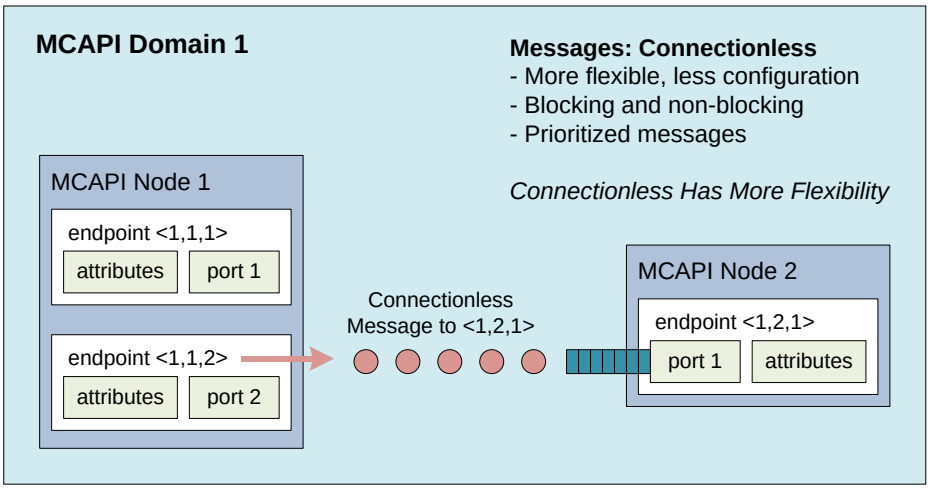


Figure 1. Connectionless Messages

2.7 Packet Channels

MCAPI packet channels, shown in Figure 2, provide a method to transmit data between endpoints by first establishing a connection, thus potentially removing or reducing the message-header and route-discovery overhead for subsequent data transmission. Packet channels are unidirectional and deliver data in a FIFO manner. The buffers are provided by the MCAPI implementation on the receive side, and by the user application on the send side. Packet channels provide per-endpoint priority. It is not allowed to send a message to a connected endpoint.

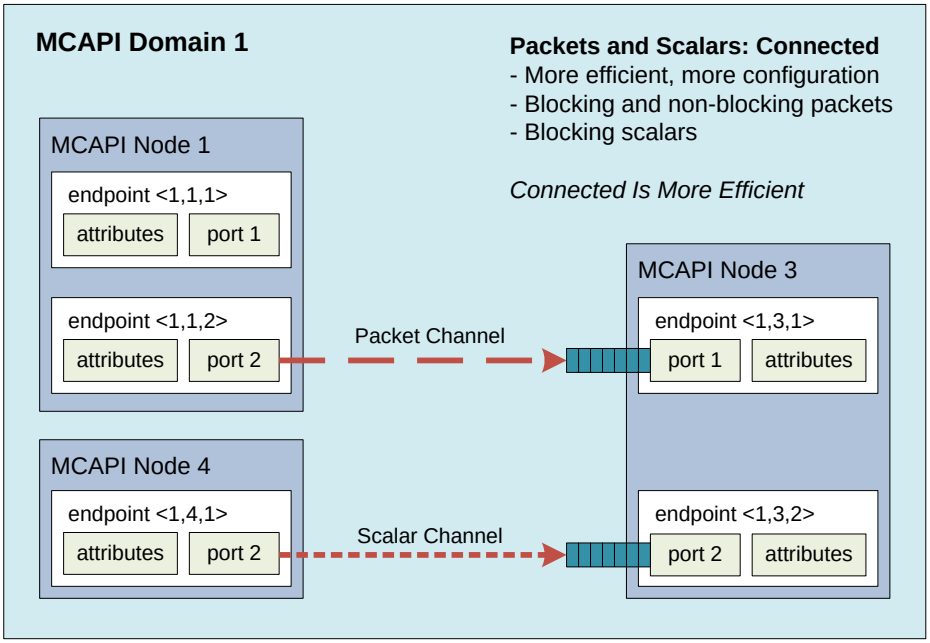


Figure 2. Connected Channels

2.8 Scalar Channels

MCAPI scalar channels, shown in Figure 2, provide a method to transmit scalars very efficiently between endpoints by first establishing a connection. Like packet channels, scalar channels are unidirectional and deliver data in a FIFO manner. The scalar functions come in 8-bit, 16-bit, 32-bit and 64-bit variants. The scalar receives must be of the same size as the scalar sends. A mismatch in size results in an error. Scalar channels provide per-endpoint priority.

2.9 Zero Copy

Zero copy means that data is passed by reference instead of physical copy. This method is useful when multiple cores have shared memory, because one core can operate on a buffer of data and simply pass a pointer to the buffer to the next core, thereby saving the time and energy of moving the data physically. Zero copy requires specific buffer management and is intended to be orthogonal to other API operations, to avoid complexity (simplifying code migration) and performance penalties for non-zero copy operations. With the addition of the `mcapi_pktchan_release_test()` function (Version 1.1) MCAPI is capable of supporting zero copy directly through the API functions, if shared memory and support in the underlying implementation is available. The application is responsible for allocation of shared memory.

2.10 Error Handling Philosophy

Error handling is a fundamental part of the specification. However, some accommodations have been made to allow for trading-off completeness for efficiency of implementation. For example, a few API functions allow implementations to optionally handle errors[2]. Consistent and efficient coding styles also govern the design of the error handling. In general, function calls include an error-code parameter used by the API function to indicate detailed status. In addition, the return value of several API functions indicates success or failure which enables efficient coding practice. A value of type, `mcapi_status_t`, indicates success or failure states of API calls. `MCAPI_NULL` is a valid return value for `mcapi_status_t` (it can be used for implementation optimization) and implementations should state when this is the case.

If a process or thread attached to a node fails, it is generally up to the application to recover from this failure. MCAPI provides timeouts, as endpoint attributes as well as for the `mcapi_wait()` and `mcapi_wait_any()` functions, and MCAPI also provides the `mcapi_cancel()` function to clear outstanding non-blocking requests at the non-failing side of the communication. It is also possible to reinitialize a failed node by first calling `mcapi_finalize()`.

[2] The `mcapi_sclchan_unit_send_*` and `mcapi_sclchan_uint_recv_*` routines define error arguments, but implementation are free to assume the routines always succeed. The message and packet channel send and receive functions can optionally report transmission errors.

2.11 Buffer Management Philosophy

MCAPI provides two methods for transferring buffers of data between endpoints: MCAPI messages and MCAPI packet channels. Both methods allow the programmer to send a buffer of data on one node and receive it on another node. The methods differ in that messaging is connectionless, allowing any node to communicate with any other node at any time, with priority. Packet channels, on the other hand, perform repeated communication via a connection between two endpoints.

The APIs for sending a buffer via messaging or packet channels are very similar. Both APIs have a send method that allows the programmer to specify a buffer of data with two parameters: (`void*, size_t`). The programmer may send any data buffer they choose. There is no requirement that the buffer be allocated by MCAPI or returned to MCAPI after the send call completes.

The messaging and packet-channel APIs differ in their handling of received buffers. The messaging API provides a "user-specified buffer" communications interface–the programmer specifies an empty buffer to be filled with incoming data on the receive side. The programmer can specify any buffer they choose and there is no requirement to allocate or release the buffer via an MCAPI call.

On the other hand, packet channels provide a "system-specified buffer" interface–the receive method returns a buffer of data at an address chosen by the runtime system. Since the receive buffer for a packet channel is allocated by the system, the programmer must return the buffer to the system by calling `mcapi_pktchan_release()`.

MCAPI data buffers may have arbitrary alignment. However, MCAPI defines two macros to help provide buffer alignment in performance-critical cases. Use of these macros is optional and can be defined with no value ("no-op"); implementers are required to handle send and receive buffers of arbitrary alignment, but can optimize for aligned buffers.

MCAPI implementations define the following two macros:

1. `MCAPI_DECL_ALIGNED` : a macro placed on static declarations in order to force the compiler to make the declared object aligned. For example:

```
mcapi_int_t MCAPI_DECL_ALIGNED my_int_to_send; /* See mcapi.h */
```

2. `MCAPI_BUF_ALIGN` : a macro that evaluates to the number of bytes to which dynamically allocated buffers should be aligned. For example:

```
memalign(MCAPI_BUF_ALIGN, sizeof(my_message));
```

MCAPI does not provide for any maximum buffer size. Applications may use any message size they choose, but implementations may specify a maximum buffer size and may fail with `MCAPI_ERR_MEM_LIMIT` if the system runs out of allocatable memory. Thus, the maximum size of a message in any particular implementation is limited by an implementation-specified size or the amount of system memory available to that implementation.

2.12 Timeout and Cancellation Philosophy

The MCAPI API provides timeout functionality for its non-blocking calls through the timeout parameters of the `mcapi_wait()` and `mcapi_wait_any()` functions. For blocking functions, implementations can optionally provide timeout capability by using endpoint attributes.

2.13 Backpressure Mechanism

In many systems, the sender and receiver must coordinate to ensure that the messages are handled without overwhelming the receiver. Without such coordination, the sender must implement elaborate recovery mechanisms in case a message is dropped by the receiver. The overhead to handle these error cases is higher than pausing for some time and continuing later. The sender handles such scenarios by inquiring about the receiver status before sending a message. The receiver returns the status based on the number of available buffers or buffer size.

The sender throttles the messages using the `mcapi_endpoint_attribute_get()` API and the `MCAPI_ATTR_NUM_RECV_BUFFERS_AVAILABLE` attribute. If required, the sender and receiver can reserve some buffers for higher-priority messages. A more sophisticated mechanism can be build on top of this API. For example, zones (green, yellow and red) can be defined using the number of available messages. The sender can use the zones to throttle the messages.

2.14 Implementation Concerns

2.14.1 Link Management

The processes for link configuration and link management are not specified by MCAPI. MCAPI does not define APIs or services (such as membership services) related to dynamic link-state detection, node or service discovery, and node- or service-state management. This is because MCAPI is designed to address multicore and multiprocessor systems in which the number of nodes and their connectivity topology is known when the MCAPI system is configured. Since the MCAPI specification does not specify any APIs that deal with network interfaces, an MCAPI implementation is free to manage links underneath the interface as it sees fit. In particular, an MCAPI implementation can restrict the topologies supported or the number of links between nodes in an MCAPI communication topology.

2.14.2 Thread-Safe Implementations

MCAPI implementations are assumed to be reentrant (thread-safe). If an MCAPI implementation is available in a threaded environment, then it needs to be thread-safe. MCAPI implementations can also be available in non-threaded environments, but the provider of such implementations needs to clearly indicate that the implementation is not thread-safe.

2.15 Potential Future Extensions

With the goals of implementing MCAPI efficiently, the APIs are kept simple with potential for adding more functionality on top of MCAPI later. Some specific areas include additional zero copy functionality, multicast, and informational functions for debugging, statistics (optimization), and status. These areas are strong candidates for future extensions, and they are briefly described in the following subsections.

2.15.1 Zero Copy

As mentioned above in Section 2.9, zero copy can be very useful for systems in which multiple nodes share memory. Zero-copy buffer management can be implemented relatively simply on top of MCAPI, and this is something to consider for future versions of MCAPI.

2.15.2 Multicast

MCAPI currently supports point-to-point communication. Communication with one sender and multiple receivers, often referred to as multi-cast, can be layered on top of MCAPI. For transports with hardware support for one-to-multiple communication, specific API functions supporting this type of communication would allow for more efficient implementations than is possible through layering.

2.15.3 Debug, Statistics and Status functions

Support functions providing information for debugging, optimization, and system status are useful in most systems. This would be a valuable addition to MCAPI, and is also worth future consideration.

2.16 Data Types

MCAPI uses predefined data types for maximum portability. The predefined MCAPI data types are defined in the following subsections. To simplify the use of multiple Multicore Association (MCA) APIs, some MCAPI data types have MCA equivalents and some MCAPI functions will have MCA-equivalent functions that can be used for multiple MCA APIs. An MCAPI implementation is not required to provide MCA-equivalent functions.

2.16.1 mcapi_domain_t

The `mcapi_domain_t` type is used for MCAPI domains. The domain ID scheme is implementation-defined. For application portability, we recommend using symbolic constants in your code. The `mcapi_domain_t` has an `mca_domain_t` equivalent.

2.16.2 mcapi_node_t

The `mcapi_node_t` type is used for MCAPI nodes. The node numbering is implementation-defined. For application portability, we recommend using symbolic constants in your code. The mcapi_node_t has an mca_node_t equivalent.

2.16.3 mcapi_port_t

The `mcapi_port_t` type is used for MCAPI ports. The port numbering is implementation-defined. For application portability, we recommend using symbolic constants in your code.

2.16.4 mcapi_endpoint_t

The `mcapi_endpoint_t` type is used for creating and managing endpoints for sending and receiving data. MCAPI routines for creating and managing endpoints are described in Section 3.3, packet channel creation is covered in Section 3.5.1, and scalar channel creation is covered in Section 3.6.1. The `mcapi_endpoint_t` type is an opaque data type whose exact definition is implementation-defined. The endpoint identifier is globally unique to an MCAPI topology.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The endpoint type must provide for topology-global uniqueness, be returnable by a function, be passable as a parameter to a function, and should allow simple arithmetic equality comparison (a == b) such as a 32- bit scalar or pointer.

2.16.5 mcapi_pktchan_recv_hndl_t

The `mcapi_pktchan_recv_hndl_t` type is used to receive packets from a connected packet channel. MCAPI routines for creating and using the `mcapi_pktchan_recv_hndl_t` type are covered in Section 3.5. The `mcapi_pktchan_recv_hndl_t` is an opaque data type whose exact definition is implementation-defined.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The handle must be passable as a parameter to a function and should allow simple arithmetic equality comparison (a == b) such as a 32- bit scalar or pointer.

2.16.6 mcapi_pktchan_send_hndl_t

The `mcapi_pktchan_send_hndl_t` type is used to send packets to a connected packet channel. MCAPI routines for creating and using the `mcapi_pktchan_send_hndl_t` type are covered in Section 3.5. The `mcapi_pktchan_send_hndl_t` is an opaque data type whose exact definition is implementation-defined.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The handle must be passable as a parameter to a function and should allow simple arithmetic equality comparison (a == b), such as a 32- bit scalar or pointer.

2.16.7 mcapi_sclchan_recv_hndl_t

The `mcapi_sclchan_recv_hndl_t` type is used to receive scalars from a connected scalar channel. MCAPI routines for creating and using the `mcapi_sclchan_recv_hndl_t` type are covered in Section 3.6. The `mcapi_sclchan_recv_hndl_t` is an opaque data type whose exact definition is implementation-defined.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The handle must be passable as a parameter to a function and should allow simple arithmetic equality comparison (a == b), such as a 32- bit scalar or pointer.

2.16.8 mcapi_sclchan_send_hndl_t

The `mcapi_sclchan_send_hndl_t` type is used to send scalars to a connected scalar channel. MCAPI routines for creating and using the `mcapi_sclchan_send_hndl_t` type are covered in Section 3.6. The `mcapi_sclchan_send_hndl_t` is an opaque data type whose exact definition is implementation-defined.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The handle must be passable as a parameter to a function and should allow simple arithmetic equality comparison (a == b), such as a 32-bit scalar or pointer.

2.16.9 Scalar Data Types

The following scalar types are used for signed and unsigned 64-, 32-, 16-, and 8-bit scalars:

- `mcapi_uint64_t`
- `mcapi_uint32_t`
- `mcapi_uint16_t`
- `mcapi_uint8_t`

2.16.10 mcapi_request_t

The `mcapi_request_t` type is used to record the state of a pending non-blocking MCAPI transaction (see Section 2.5.4). Non-blocking MCAPI routines exist for message send and receive (see Section 3.4) and packet send and receive (see Section 3.5). The MCAPI request can only be used by the node it was created on. The `mcapi_request_t` has an `mca_request_t` equivalent.

NOTE: Do not attempt to examine the contents of this data type, as this can result in non-portable application code.

Implementation advice: The request should allow simple arithmetic equality comparison (a == b), such as a 32-bit scalar or pointer.

2.16.11 mcapi_status_t

The `mcapi_status_t` type is an enumerated type used to record the result of an MCAPI API call. If a status can be returned by an API call, the associated MCAPI API call will allow a `mcapi_status_t` to be passed by reference. The API call will fill in the status code and the API user may examine the `mcapi_status_t` variable to determine the result of the call. The function return values are valid only when `mcapi_status` returns `MCAPI_SUCCESS`. The `mcapi_status_t` has an `mca_status_t` equivalent.

2.16.12 mcapi_timeout_t

The `mcapi_timeout_t` type is an unsigned scalar type used to indicate the duration that an `mcapi_wait()` or `mcapi_wait_any()` API call will block before reporting a timeout. The units of the `mcapi_timeout_t` data type are implementation-defined because mechanisms for time keeping vary from system to system. Applications should not rely on this feature for satisfaction of real-time constraints because its use will not guarantee application portability across MCAPI implementations.

The `mcapi_timeout_t` data type is intended primarily to allow for error detection and recovery, and an application developer must take appropriate action if it is used for any other purpose. The `mcapi_timeout_t` has an `mca_timeout_t` equivalent. A value of `MCAPI_TIMEOUT_IMMEDIATE (0)` for the timeout parameter indicates that the function will return without blocking, indicating failure or success; a value of `MCAPI_INFINITE (~0)` for the timeout parameter indicates no timeout is requested, i.e. the function will block until it is unblocked because of failure or success.

2.16.13 Endpoint Attributes

MCAPI endpoint attributes provide access to endpoint characteristics, and state. See the header files for detailed information. Implementations may designate endpoint attributes as read-only. Some endpoint attributes have to be compatible for a successful channel, as noted below.

2.16.13.1 mcapi_endp_attr_max_payload_size_t

This attribute defines the maximum payload size. This is a channel compatibility attribute; a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.

2.16.13.2 mcapi_endp_attr_buffer_type_t

This attribute defines the endpoint buffer type, at the present only FIFO type exists. It means that the order of transmission is FIFO for channels and FIFO per priority level for messages. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.

Buffer types: `MCAPI_ENDP_ATTR_FIFO_BUFFER` Default

2.16.13.3 mcapi_endp_attr_memory_type_t

This attribute defines the memory's locality, whether local, shared, or remote. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints.

Memory locality:

`MCAPI_ENDP_ATTR_LOCAL_MEMORY` Default
`MCAPI_ENDP_ATTR_SHARED_MEMORY`
`MCAPI_ENDP_ATTR_REMOTE_MEMORY`

2.16.13.4 mcapi_endp_attr_num_priorities_t

This attribute defines the number of endpoint priorities. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.

2.16.13.5 mcapi_endp_attr_priority_t

This attribute defines the endpoint priority, applied to a channel at the time the channel is connected. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. A lower number means higher priority. A value of `MCAPI_MAX_PRORITY (0)` denotes the highest priority.

2.16.13.6 mcapi_endp_attr_num_send_buffers_t

This attribute contains the number of send buffers at the current endpoint priority level. Default value is implementation-defined.

2.16.13.7 mcapi_endp_attr_num_recv_buffers_t

This attribute contains the number of receive buffers available. This can, for example, be used for throttling. Implementation-defined default value.

2.16.13.8 **mcapi_endp_attr_status_t**

This attribute contains endpoint status flags. Flags are used to query the status of an endpoint, e.g. if it is connected, and if so what type of channel, direction, etc.

Note: The lower 16 bits are defined in `mcapi.h` whereas the upper 16 bits are reserved for implementation-specific purposes and, if used, must be defined in `implementation_spec.h`. It is therefore recommended that the upper 16 bits be masked off at the application level.

```
0x00000000
      ----  mcapi.h
  ----            implementation_spec.h
```

```
Default = 0x00000000
```

Standard status flags:

```
MCAPI_ENDP_ATTR_STATUS_CONNECTED      /* The endpoint is one end of a
                                      connected channel*/
MCAPI_ENDP_ATTR_STATUS_OPEN           /* A channels is open on this
                                      endpoint*/
MCAPI_ENDP_ATTR_STATUS_OPEN_PENDING   /* A channel open is pending */
MCAPI_ENDP_ATTR_STATUS_CLOSE_PENDING  /* A channel close is pending */
MCAPI_ENDP_ATTR_STATUS_PKTCHAN        /* Packet channel */
MCAPI_ENDP_ATTR_STATUS_SCLCHAN        /* Scalar channel */
MCAPI_ENDP_ATTR_STATUS_SEND           /* Send side */
MCAPI_ENDP_ATTR_STATUS_RECEIVE        /* Receive side */
```

2.16.13.9 **mcapi_endp_attr_timeout_t**

This attribute contains the timeout value for blocking send and receive functions. A value of `MCAPI_TIMEOUT_IMMEDIATE` (or 0) means that the function will return "immediately", with success or failure. `MCAPI_TIMEOUT_INFINITE` means that the function will block until it completes with success or failure. Default = `MCAPI_TIMEOUT_INFINITE`.

2.16.14 **Other MCAPI Data Types**

MCAPI also defines its own integer, Boolean and other types, some of which have MCA equivalents. See the header files on page 141 of this document for specifics on these data types.

2.16.15 Error and Status Codes

Status code	Description
MCAPI_SUCCESS	Indicates operation was successful
MCAPI_PENDING	Indicates operation is pending without errors
MCAPI_TIMEOUT	The operation timed out
MCAPI_ERR_PARAMETER	Incorrect parameter
MCAPI_ERR_DOMAIN_INVALID	The parameter is not a valid domain
MCAPI_ERR_NODE_INVALID	The parameter is not a valid node
MCAPI_ERR_NODE_INITFAILED	The MCAPI node could not be initialized
MCAPI_ERR_NODE_INITIALIZED	MCAPI node is already initialized
MCAPI_ERR_NODE_NOTINIT	The MCAPI node is not initialized
MCAPI_ERR_NODE_FINALFAILED	The MCAPI could not be finalized
MCAPI_ERR_PORT_INVALID	The parameter is not a valid port
MCAPI_ERR_ENDP_INVALID	The parameter is not a valid endpoint descriptor
MCAPI_ERR_ENDP_EXISTS	The endpoint is already created
MCAPI_ERR_ENDP_GET_LIMIT	The endpoint get reference count is too high
MCAPI_ERR_ENDP_NOTOWNER	An endpoint can only be deleted by its creator
MCAPI_ERR_ENDP_REMOTE	Certain operations are only allowed on the node local endpoints
MCAPI_ERR_ATTR_INCOMPATIBLE	Connection of endpoints with incompatible attributes not allowed
MCAPI_ERR_ATTR_SIZE	Incorrect attribute size
MCAPI_ERR_ATTR_NUM	Incorrect attribute number
MCAPI_ERR_ATTR_VALUE	Incorrect attribute vale
MCAPI_ERR_ATTR_NOTSUPPORTED	Attribute not supported by the implementation
MCAPI_ERR_ATTR_READONLY	Attribute is read-only
MCAPI_ERR_MSG_SIZE	The message size exceeds the maximum size allowed by the MCAPI implementation
MCAPI_ERR_MSG_TRUNCATED	The message size exceeds the buffer size
MCAPI_ERR_CHAN_OPEN	A channel is open, certain operations are not allowed
MCAPI_ERR_CHAN_TYPE	Attempt to open a packet/scalar channel on an endpoint that has been connected with a different channel type
MCAPI_ERR_CHAN_DIRECTION	Attempt to open a send handle on a port that was connected as a receiver, or vice versa
MCAPI_ERR_CHAN_CONNECTED	A channel connection has already been established for one or both of the specified endpoints
MCAPI_ERR_CHAN_OPENPENDING	An open request is pending
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_CHAN_NOTOPEN	The channel is not open (cannot be closed)
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle
MCAPI_ERR_PKT_SIZE	The packet size exceeds the maximum size allowed by the MCAPI implementation
MCAPI_ERR_TRANSMISSION	Transmission failure
MCAPI_ERR_PRIORITY	Incorrect priority level
MCAPI_ERR_BUF_INVALID	Not a valid buffer descriptor
MCAPI_ERR_MEM_LIMIT	Out of memory
MCAPI_ERR_REQUEST_INVALID	Argument is not a valid request handle

Status code	Description
MCAPI_ERR_REQUEST_LIMIT	Out of request handles
MCAPI_ERR_REQUEST_CANCELLED	The request was already canceled
MCAPI_ERR_WAIT_PENDING	A wait is pending
MCAPI_ERR_GENERAL	To be used by implementations for error conditions not covered by the other status codes
MCAPI_STATUSCODE_END	This should always be last

2.16.16 API Parameter Readability

MCAPI_IN and MCAPI_OUT are used only for clarity to distinguish between input and output parameters.

2.17 What is New in MCAPI 2.015

The changes from Version 1.0 to Version 2.000 are in the areas of improved API consistency, bug fixes, enhancements, and added functionality, as described below.

MCAPI Version 2.015 is functionally identical to MCAPI Version 2.000. The only differences are clarifications and grammatical improvements in the text.

2.17.1 MCAPI to MCA types

Some MCAPI data types and definitions were promoted to MCA data types and definitions to simplify interoperability between MCAPI and other MCA standards and to allow implementations to provide MCA versions of some functions that could be used for multiple MCA standards. For this purpose, an mca.h header file was added.

2.17.2 Domains

An MCAPI domain is comprised of one or more MCAPI nodes in a multicore topology and used for routing purposes. The scope of a domain is implementation-defined and its scope could, for example, be a single chip with multiple cores or multiple processor chips on a board. The domain ID is specified once at node initialization. A domain can contain multiple nodes. Some example uses for domains are topologies that may change dynamically, topologies that include non-MCAPI sub-topologies, and topologies that require separation between different transports or have open and secure areas.

2.17.3 Endpoint Attributes

A few attributes were added. For details see the ATTRIBUTES subsections in Section 3.3. The first 64 endpoint attributes (numbers and data types) are reserved for the Multicore Association (MCA). Vendors desiring to add vendor-specific endpoint attributes in their implementations can request, from the MCA, to be assigned ranges of 32 vendor-specific endpoint attribute numbers. Attribute ranges are defined in mca.h.

2.17.4 Header Files

The header files have been restructured to better separate standard and implementation-specific definitions.

2.17.4.1 **New Functionality and Functions**

2.17.4.2 **Parameter types**

The parameters `mcapi_port_t` and `mcapi_timeout_t` (= `mca_tiemout_t`) were changed from `int` to `unsigned int`, to provide a longer range of meaningful values.

2.17.4.3 **Initialization Parameters and Information**

Initialization parameters were added to allow implementations to configure the MCAPI at runtime. A parameter was also added to allow implementations to provide information about the MCAPI runtime with both MCAPI-specified and implementation-specific information. For specifics on MCAPI information, see below. The topology information can, for example, be used for basic initial system discovery by using the `number_of_domains` and `number_of_nodes` parameters to establish communications for more thorough discovery.

2.17.4.3.1 *mcapi_node_attributes_t*

Node initialization parameters defined by MCAPI.

MCAPI -defined node attributes:

`MCAPI_NODE_ATTR_TYPE_REGULAR` The node is regular. Default.

2.17.4.3.2 *mcapi_param_t*

Initialization parameters will vary by implementation, and may include specification of the amount of resources to be used for a specific implementation or configuration, such as the maximum number of outstanding requests. This means that each implementation must provide documentation and/or sample code for the initialization. Some initialization parameters may be standardized in future versions of MCAPI.

2.17.4.3.3 *mcapi_info_t*

The informational parameters include MCAPI-specified information as outlined below, as well as implementation-specific information. Implementation-specific information must be documented by the implementer.

MCAPI-defined initialization information:

`mcapi_version`	MCAPI version, the three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.
`organization_id`	Implementation vendor or organization ID. Assigned by MCA.
`implementation_version`	Implementation version, represented by a 32-bit scalar. The specific format is implementation-defined.
`number_of_domains`	Number of domains in the topology.
`number_of_nodes`	Number of nodes in the domain. This can be used for basic per-domain topology discovery. Note: This information may not be available in all implementations. The functionality must be documented.
`number_of_ports`	Number of available ports on the local node.

2.17.4.4 **MCAPI_domains**

An MCAPI domain is comprised of one or more MCAPI nodes in a multicore topology and used for routing purposes. The scope of a domain is implementation-defined and could, for example, be a single chip with multiple cores or multiple processor chips on a board. The domain ID is specified once at node initialization. A domain can contain multiple nodes. Some example uses for domains are topologies that may change dynamically, topologies that include non-MCAPI sub-topologies, or topologies that have open and secure areas.

2.17.4.5 **mcapi_domain_id_get()**

This function was added to allow the application to find its MCAPI domain ID. Other functions affected are: `mcapi_initialize(), mcapi_endpoint_get_i() and mcapi_endpoint_get().`

2.17.4.6 **mcapi_node_id_get()**

The return value was changed from mcapi_uint_t to mcapi_node_t for mca data type alignment purposes.

2.17.4.7 **mcapi_endpoint_get() Functionality**

To avoid the potential for blocking indefinitely on this function, a timeout parameter was added.

2.17.4.8 **mcapi_node_init_attribute()**

This function was added to allow initialization of a node's attribute structure for setting non-default node attributes to be used by `mcapi_initialize()`.

2.17.4.9 **mcapi_node_set_attribute()**

This function was added to allow setting non-default node attributes to be used by `mcapi_initialize()`.

2.17.4.10 **mcapi_node_get_attribute()**

This function was added to allow the application to query other nodes about their attributes, e.g. type of node, characteristics, etc. This functionality takes basic topology discovery a step further and simplifies applying MCAPI to a multitude of diverse multicore platforms.

2.17.4.11 **mcapi_pktchan_release_test()**

This function was added to allow the sender side of a packet channel to find out if a buffer has been released by the receiver for reuse. This may, for example, be useful for zero copy operations.

2.17.4.12 **mcapi_wait()**

The order of the timeout and mcapi_status parameters was changed for consistency.

2.17.4.13 **mcapi_wait_any()**

The order of the timeout and mcapi_status parameters was changed for consistency. The mcapi_request_t** paramater was change to to mcapi_request_t*, to simplify usage. The return value was changed return from mcapi_int_t to mcapi_uint_t, as a negative value is not meaningful in the context of this function.

2.17.4.14 **mcapi_display_status()**

This function was added to enable displaying an MCAPI status code in text format, for convenience.

2.17.5 **API Consistency**

2.17.5.1 **API Function Names**

Some API function names were modified to adhere to a consistent nomenclature. The nomenclature is `mcapi_<functional area>_<action>`, e.g. `mcapi_msg_send()`.

The following functions were name changed:

- mcapi_node_id_get()
- mcapi_endpoint_create()
- mcapi_endpoint_delete()
- mcapi_endpoint_get_i()
- mcapi_endpoint_get()
- mcapi_endpoint_get_attribute()
- mcapi_endpoint_set_attribute()
- mcapi_pktchan_connect_i()
- mcapi_pktchan_recv_open_i()
- mcapi_pktchan_send_open_i()
- mcapi_pktchan_release()
- mcapi_pktchan_recv_close_i()
- mcapi_pktchan_send_close_i()
- mcapi_sclchan_connect_i()
- mcapi_sclchan_recv_open_i()
- mcapi_sclchan_send_open_i()

2.17.5.2 **Error and Status Codes**

Error and status codes were modified for consistency, and others were added to allow for more precise error and status reporting.

2.17.6 **Definitions and Constants**

Some definitions and constants have been more clearly defined and specified.

2.17.7 **Clarifications**

Clarifications were added throughout the document to improve the understanding of the MCAPI specification.

2.18 **MCAPI 2.015 to 1.0 Backwards Compatibility**

To address backwards compatibility as much as possible, a new header file `mcapi_v1000_to_v2000.h` and a compatibility function were added. A few notes on the backwards compatibility follow.

2.18.1 Affected Functions

2.18.1.1 mcapi_initialize()

The `mcapi_initialize()` function has the additional parameters `domain_id`, `mcapi_node_attributes`, and `init_parameters` and modified `mcapi_info` parameter. An `mcapi_1000_intitialize()` function is added in the header file showing how to map to the v 2.000 `mcapi_initialize()` function. MCAPI 1.0 nodes will always be in domain 0.

2.18.1.2 mcapi_endpoint_get_i() and mcapi_endpoint_get()

The `mcapi_endpoint_get_i()` and `mcapi_endpoint_get()` functions have the additional `domain_id` parameter and `mcapi_endpoint_get()` a new timeout parameter. `mcapi_1000_endpoint_get_i()` and `mcapi_1000_endpoint_get()` functions are added in the header file showing how to map to the v 2.000 functions. MCAPI 1.0 endpoints will always be in domain 0.

2.18.1.3 Modified Function Names and Status or Error Codes

Functions with modified names and unchanged parameters are handled with defines. Modified status and error codes are handled with defines.

3. MCAPI API

The MCAPI API is divided into seven major parts:

- General
- Endpoints
- Messages
- Packet channels
- Scalar channels
- Non-blocking operations
- Support functions

The following sections enumerate the API calls for each of these major parts.

3.1 Conventions

`MCAPI_IN` and `MCAPI_OUT` are used to distinguish between input and output parameters.

3.2 General

This section describes initialization and introspection functions. All applications wishing to use MCAPI functionality must use the initialization and finalization routines. Following initialization, the introspection functions can provide important information to MCAPI-based applications.

3.2.1 MCAPI_INITIALIZE

NAME

mcapi_initialize

SYNOPSIS

```
#include <mcapi.h>

void mcapi_initialize(
   MCAPI_IN mcapi_domain_t domain_id,
   MCAPI_IN mcapi_node_t node_id,
   MCAPI_IN mcapi_node_attributes_t* mcapi_node_attributes,
   MCAPI_IN mcapi_param_t* mcapi_parameters,
   MCAPI_OUT mcapi_info_t* mcapi_info,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

mcapi_initialize() initializes the MCAPI environment on a given MCAPI node in a given MCAPI domain. It has to be called by each node using MCAPI. mcapi_node_attributes is used to pass MCAPI-defined initialization parameters. mcapi_parameters is used to pass implementation specific initialization parameters. mcapi_info is used to obtain information from the MCAPI implementation, including MCAPI standardized information, see below and the header files and vendor specific implementation information. A node is a process, a thread, or a processor (or core) with an independent program counter running a piece of code. In other words, an MCAPI node is an independent thread of control. An MCAPI node can call mcapi_initialize() once per node, and it is an error to call mcapi_initialize() multiple times from a given node, unless mcapi_finalize() is called in between. A given MCAPI implementation will specify what is a node (i.e., what thread of control – process, thread, or other -- is a node) in that implementation. A thread and process are just two examples of threads of control, and there could be others.

Initialization parameters will vary by implementation, and may include specifications of the amount of resources to be used for a specific implementation, such as the maximum number of outstanding requests, etc. An mcapi_node_attributes structure is passed in by reference. A NULL value for the mcapi_node_attributes pointer indicates that default values should be set.

MCAPI-defined node attributes:

MCAPI_NODE_ATTR_TYPE_REGULAR The node is regular. Default.

The informational parameters include MCAPI-specified information as outlined below, as well as implementation specific information. Implementation specific information shall be documented by the implementer.

MCAPI-defined initialization information:

mcapi_version	MCAPI version, the three last (rightmost) hex digits are the minor number and those left of minor the major number.
organization_id	Implementation vendor/organization ID.
implementation_version	Vendor version, the three last (rightmost) hex digits are the minor number and those left of minor the major number.

implementation_version	Vendor version, the three last (rightmost) hex digits are the minor number and those left of minor the major number.
number_of_domains	Number of domains in the topology.
number_of_nodes	Number of nodes in the domain, can be used for basic per domain topology discovery.
number_of_ports	Number of ports on the local node

RETURN VALUE

On success, *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_INITFAILED	The MCAPI environment could not be initialized.
MCAPI_ERR_NODE_INITIALIZED	The MCAPI environment has already been initialized.
MCAPI_ERR_NODE_INVALID	The parameter is not a valid node.
MCAPI_ERR_DOMAIN_INVALID	The parameter is not a valid domain.
MCAPI_ERR_PARAMETER	Incorrect mcapi_parameters or mcapi_info parameter. Note, mcapi_parameters can be a NULL pointer, if the MCAPI version is 1.000, for backwards compatibility purposes.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_finalize()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.2.2 MCAPI_FINALIZE

NAME

`mcapi_finalize`

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

`mcapi_finalize()` finalizes the MCAPI environment on the local MCAPI node. It has to be called by each node using MCAPI, to un-initialize the node. It is an error to call `mcapi_finalize()` without first calling `mcapi_initialize()`. An MCAPI node can call `mcapi_finalize()` once for each call to `mcapi_initialize()`, but it is an error to call `mcapi_finalize()` multiple times from a given node unless `mcapi_initialize()` has been called prior to each `mcapi_finalize()` call. Pending data and outstanding requests will be discarded when `mcapi_finalize()` is called. Implementations must document other specific functionality of `mcapi_finalize()`, if applicable.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_NODE_FINALFAILED`	The MCAPI environment could not be finalized.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

`mcapi_initialize()`

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.2.3 MCAPI_DOMAIN_ID_GET

NAME

`mcapi_domain_id_get`

SYNOPSIS

```
#include <mcapi.h>

mcapi_domain_t mcapi_domain_id_get(
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Returns the domain ID associated with the local node.

RETURN VALUE

On success the `domain_id` is returned, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below and the return value is set to `MCAPI_DOMAIN_INVALID`.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.2.4 MCAPI_NODE_ID_GET

NAME

mcapi_node_id_get

SYNOPSIS

```
#include <mcapi.h>

mcapi_node_t mcapi_node_id_get(
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Returns the node ID associated with the local node.

RETURN VALUE

On success the node_id is returned, *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below and the return value is set to MCAPI_NODE_INVALID.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.2.5 MCAPI_NODE_INIT_ATTRIBUTES

NAME

`mcapi_node_init_attributes`

SYNOPSIS

```
void mcapi_node_init_attributes(
   MCAPI_OUT mcapi_node_attributes_t* mcapi_node_attributes,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

This function initializes the values of an `mcapi_node_attributes_t` structure. For non-default behavior this function should be called prior to calling `mcapi_node_set_attribute()`. `mcapi_node_set_attribute()` is then used to change any default values prior to calling `mcapi_initialize()`.

MCAPI-defined node attributes:

`MCAPI_NODE_ATTR_TYPE_REGULAR` The node is regular. Default.

RETURN VALUE

On success `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.2.6 MCAPI_NODE_SET_ATTRIBUTE

NAME

mcapi_node_set_attribute

SYNOPSIS

```
void mcapi_node_set_attribute(
   MCAPI_OUT mcapi_node_attributes_t* mcapi_node_attributes,
   MCAPI_IN mcapi_uint_t attribute_num,
   MCAPI_IN void* attribute,
   MCAPI_IN size_t attribute_size,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

This function is used to change default values of an mcapi_node_attributes_t data structure prior to calling mcapi_initialize(). This is a blocking function. Calls to this function have no effect on node attributes once the MCAPI has been initialized. The purpose of this function is to define node specific characteristics, and in this MCAPI version has only attribute. The node attributes are expected to be expanded in future versions.

MCAPI-defined node attributes:

MCAPI_NODE_ATTR_TYPE_REGULAR The node is regular. Default.

RETURN VALUE

On success *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_ATTR_NUM	Unknown attribute number.
MCAPI_ERR_ATTR_VALUE	Incorrect attribute value.
MCAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MCAPI_ERR_ATTR_NOTSUPPORTED	Attribute not supported by the implementation.
MCAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MCAPI_ERR_PARAMETER	Incorrect mcapi_node_attributes or attribute parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.2.7 MCAPI_NODE_GET_ATTRIBUTE

NAME

mcapi_node_get_attribute – Get node attributes from a remote node.

SYNOPSIS

```
#include <mcapi.h>

void mcapi_node_get_attribute(
   MCAPI_IN mcapi_domain_t domain_id,
   MCAPI_IN mcapi_node_t node_id,
   MCAPI_IN mcapi_uint_t attribute_num,
   MCAPI_OUT void* attribute,
   MCAPI_IN size_t attribute_size,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

mcapi_node_get_attribute() allows the application to query other nodes about their attributes, e.g. type of node, characteristics, etc. domain_id and node_id specify the node being queried, attribute_num indicates which one of the node attributes is being referenced. This is a blocking function. attribute points to a structure or scalar to be filled with the value of the attribute specified by attribute_num. attribute_size is the size in bytes of the attribute. See Section 2.2 and header files (Section 7) for a description of attributes. The mcapi_node_get_attribute() function returns the requested attribute value by reference.

MCAPI-defined node attributes:

MCAPI_NODE_ATTR_TYPE_REGULAR The node is regular. Default.

RETURN VALUE

On success, *attribute is filled with the requested attribute and *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to an error code and *attribute is not modified.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The local node is not initialized.
MCAPI_ERR_DOMAIN_INVALID	The parameter is not a valid domain.
MCAPI_ERR_NODE_INVALID	The parameter is not a valid node.
MCAPI_ERR_ATTR_NUM	Unknown attribute number.
MCAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MCAPI_ERR_ATTR_NOTSUPPORTED	Attribute not supported by the implementation.
MCAPI_ERR_PARAMETER	Incorrect attribute parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.3 Endpoints

This section describes API functions that create, delete, get and modify endpoints.

3.3.1 MCAPI_ENDPOINT_CREATE

NAME

`mcapi_endpoint_create`

SYNOPSIS

```
#include <mcapi.h>

mcapi_endpoint_t mcapi_endpoint_create(
   MCAPI_IN mcapi_port_t port_id,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_endpoint_create()` is used to create an endpoint on the local node with the specified port_id. A `port_id` of `MCAPI_PORT_ANY` is used to request the next available endpoint on the local node.

MCAPI supports a simple static naming scheme to create endpoints based on global tuple names, <domain_id, node_id, port_id>. Other nodes can access the created endpoint by calling `mcapi_endpoint_get()` and specifying the appropriate domain, node and port id.

Static naming allows the programmer to define an MCAPI communication topology at compile time. This facilitates simple initialization. Section 5.1 illustrates an example of initialization and bootstrapping using static naming. Creating endpoints using `MCAPI_PORT_ANY` provides a convenient method to create endpoints without having to specify the `port_id`.

RETURN VALUE

On success, an endpoint is returned and *`mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` (or 0) is returned and `*mcapi_status` is set to the appropriate error defined below. `MCAPI_NULL` (or 0) could be a valid endpoint value so status has to be checked to ensure correctness.

ERRORS

`MCAPI_ERR_PORT_INVALID`	The parameter is not a valid port. There may be no more available ports or the port may be reserved.
`MCAPI_ERR_ENDP_EXISTS`	The endpoint is already created.
`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The node ID can only be set using the `mcapi_intialize()` function.

SEE ALSO

`mcapi_initialize()`

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.3.2 MCAPI_ENDPOINT_GET_I

NAME

mcapi_endpoint_get_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_endpoint_get_i(
   MCAPI_IN mcapi_domain_t domain_id,
   MCAPI_IN mcapi_node_t node_id,
   MCAPI_IN mcapi_port_t port_id,
   MCAPI_OUT mcapi_endpoint_t* endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

mcapi_endpoint_get_i() allows other nodes ("third parties") to get the endpoint identifier for the endpoint associated with a global tuple name <domain_id, node_id, port_id>. This function is non-blocking and will return immediately.

RETURN VALUE

On success, *mcapi_status is set to MCAPI_SUCCESS if completed and MCAPI_PENDING if not yet completed. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_PORT_INVALID	The parameter is not a valid port. This error also covers endpoints without ports.
MCAPI_ERR_NODE_INVALID	The parameter is not a valid node.
MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_DOMAIN_INVALID	The parameter is not a valid domain.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect endpoint or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE: Use the mcapi_test(), mcapi_wait() and mcapi_wait_any() functions to query the status of and mcapi_cancel() function to cancel the operation.

SEE ALSO

mcapi_node_id_get()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.3.3 MCAPI_ENDPOINT_GET

NAME

mcapi_endpoint_get

SYNOPSIS

```
#include <mcapi.h>

mcapi_endpoint_t mcapi_endpoint_get(
   MCAPI_IN mcapi_domain_t domain_id,
   MCAPI_IN mcapi_node_t node_id,
   MCAPI_IN mcapi_port_t port_id,
   MCAPI_IN mcapi_timeout_t timeout,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

`mcapi_endpoint_get()` allows other nodes ("third parties") to get the endpoint identifier for the endpoint associated with a global tuple name `<domain_id, node_id, port_id>`. This function will block until the specified remote endpoint has been created via the `mcapi_endpoint_create()` call or a timeout is reached. A timeout value of `MCAPI_TIMEOUT_INFINITE` would cause this function to block until completion (success or failure).

RETURN VALUE

On success, an endpoint is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL (or 0)` is returned and `*mcapi_status` is set to the appropriate error defined below. `MCAPI_NULL (or 0)` could be a valid endpoint value so status has to be checked to ensure correctness.

ERRORS

MCAPI_ERR_PORT_INVALID	The parameter is not a valid port. This error also covers endpoints without ports (routing nodes).
MCAPI_ERR_NODE_INVALID	The parameter is not a valid node.
MCAPI_ERR_DOMAIN_INVALID	The parameter is not a valid domain.
MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_TIMEOUT	The operation timed out.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.3.4 MCAPI_ENDPOINT_DELETE

NAME

`mcapi_endpoint_delete`

SYNOPSIS

```
#include <mcapi.h>

void mcapi_endpoint_delete(
   MCAPI_IN mcapi_endpoint_t endpoint,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Deletes an MCAPI endpoint. Pending messages are discarded. If an endpoint has been connected to a packet or scalar channel, the appropriate close method must be called before deleting the endpoint. Delete is a blocking operation. Since the connection is closed before deleting the endpoint, the delete method does not require any cross-process synchronization and is guaranteed to return in a timely manner (operation will return without having to block on any IPC to any remote nodes). It is an error to attempt to delete an endpoint that has not been closed. Only the node that created an endpoint can delete it. An endpoint that has an open pending but is not yet connected can be deleted.

RETURN VALUE

On success, `*mcapi_status` is set to MCAPI_SUCCESS. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_ENDP_INVALID`	Argument is not a valid endpoint descriptor.
`MCAPI_ERR_CHAN_CONNECTED`	A channel is connected, deletion is not allowed
`MCAPI_ERR_ENDP_NOTOWNER`	An endpoint can only be deleted by its creator.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_endpoint_create()

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.3.5 MCAPI_ENDPOINT_GET_ATTRIBUTE

NAME

mcapi_endpoint_get_attribute

SYNOPSIS

```
#include <mcapi.h>

void mcapi_endpoint_get_attribute(
   MCAPI_IN mcapi_endpoint_t endpoint,
   MCAPI_IN mcapi_uint_t attribute_num,
   MCAPI_OUT void* attribute,
   MCAPI_IN size_t attribute_size,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

mcapi_endpoint_get_attribute() allows the programmer to query endpoint attributes on both node local en remote endpoints. This is a blocking function. attribute_num indicates which one of the endpoint attributes is being referenced. attribute points to a structure or scalar to be filled with the value of the attribute specified by attribute_num. attribute_size is the size in bytes of the memory pointed to by attribute. See Section 2.2 and header files (Section 7) for a description of attributes

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (whether attributes are compatible or not is implementation-defined). It is also an error to attempt to change the attributes of endpoints that are connected.

RETURN VALUE

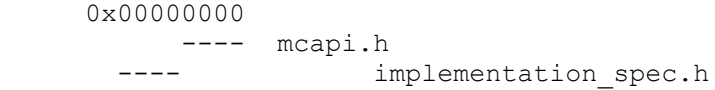
On success, *attribute is filled with the requested attribute and *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to an error code and *attribute is not modified.

ATTRIBUTES

mcapi_endp_attr_max_payload_size_t	This attribute defines the maximum payload size. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.
mcapi_endp_attr_buffer_type_t	This attribute defines the endpoint buffer type, at the present only FIFO type exists. It means that the order of transmission is FIFO for channels and FIFO per priority level for messages. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined. Buffer types: MCAPI_ENDP_ATTR_FIFO_BUFFER (Default).

mcapi_endp_attr_memory_type_t	This attribute defines the memory type both the memory's locality, local, shared and remote. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Memory locality: MCAPI_ENDP_ATTR_LOCAL_MEMORY (Default) MCAPI_ENDP_ATTR_SHARED_MEMORY MCAPI_ENDP_ATTR_REMOTE_MEMORY
mcapi_endp_attr_num_priorities_t	This attribute defines the number of endpoint priorities. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.
mcapi_endp_attr_priority_t	This attribute defines the endpoint priority, applied to a channel at the time the channel is connected. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. A lower number means higher priority. A value of MCAPI_MAX_PRORITY (0) denotes the highest priority.
mcapi_endp_attr_num_send_buffers_t	This attribute contains the number of send buffers at the current endpoint priority level. Default value is implementation-defined.
mcapi_endp_attr_num_recv_buffers_t	This attribute contains the number of receive buffers available. This can for example be used for throttling. Implementation-defined default value
mcapi_endp_attr_status_t	This attribute contains endpoint status flags. Flags are used to query the status of an endpoint, e.g. if it is connected and if so what type of channel, direction, etc. See Note A and STANDARD STATUS FLAGS, below this table.
mcapi_endp_attr_timeout_t	This attribute contains the timeout value for blocking send and receive functions. a value of MCAPI_TIMEOUT_IMMEDIATE (or 0) means that the function will return "immediately", with success or failure. MCAPI_TIMEOUT_INFINITE means that the function will block until it completes with success or failure. Default = MCAPI_TIMEOUT_INFINITE.

Note A: The lower 16 bits are defined in mcapi.h whereas the upper 16 bits are reserved for implementation specific purposes and if used must be defined in implementation_spec.h. It is therefore recommended that the upper 16 bits be masked off at the application level.



Default = 0x00000000

STANDARD STATUS FLAGS

MCAPI_ENDP_ATTR_STATUS_CONNECTED	The endpoint is one end of a connected channel.
MCAPI_ENDP_ATTR_STATUS_OPEN	A channels is open on this endpoint.
MCAPI_ENDP_ATTR_STATUS_OPEN_PENDING	A channel open is pending.
MCAPI_ENDP_ATTR_STATUS_CLOSE_PENDING	A channel close is pending.
MCAPI_ENDP_ATTR_STATUS_PKTCHAN	Packet channel.
MCAPI_ENDP_ATTR_STATUS_SCLCHAN	Scalar channel.
MCAPI_ENDP_ATTR_STATUS_SEND	Send side.
MCAPI_ENDP_ATTR_STATUS_RECEIVE	Receive side.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not an endpoint descriptor.
MCAPI_ERR_ATTR_NUM	Unknown attribute number.
MCAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MCAPI_ERR_ATTR_NOTSUPPORTED	Attribute not supported by the implementation.
MCAPI_ERR_PARAMETER	Incorrect attribute parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_endpoint_set_attribute()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.3.6 MCAPI_ENDPOINT_SET_ATTRIBUTE

NAME

mcapi_endpoint_set_attribute

SYNOPSIS

```
#include <mcapi.h>

void mcapi_endpoint_set_attribute(
   MCAPI_IN mcapi_endpoint_t endpoint,
   MCAPI_IN mcapi_uint_t attribute_num,
   MCAPI_IN const void* attribute,
   MCAPI_IN size_t attribute_size,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

mcapi_endpoint_set_attribute() allows the programmer to assign endpoint attributes on the local node. attribute_num indicates which one of the endpoint attributes is being referenced. attribute points to a structure or scalar to be filled with the value of the attribute specified by attribute_num. attribute_size is the size in bytes of the memory pointed to by attribute. See Section 2.3 and mcapi.h for a description of attributes. Endpoint attributes can be set only on the local node.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (whether attributes are compatible or not is implementation-defined). It is also an error to attempt to change the attributes of endpoints that are connected.

ATTRIBUTES

mcapi_endp_attr_max_payload_size_t	This attribute defines the maximum payload size. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.
mcapi_endp_attr_buffer_type_t	This attribute defines the endpoint buffer type, at the present only FIFO type exists. It means that the order of transmission is FIFO for channels and FIFO per priority level for messages. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined. Buffer types: MCAPI_ENDP_ATTR_FIFO_BUFFER (Default).

mcapi_endp_attr_memory_type_t	This attribute defines the memory type both the memory's locality, local, shared and remote. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Memory locality: MCAPI_ENDP_ATTR_LOCAL_MEMORY (Default) MCAPI_ENDP_ATTR_SHARED_MEMORY MCAPI_ENDP_ATTR_REMOTE_MEMORY
mcapi_endp_attr_num_priorities_t	This attribute defines the number of endpoint priorities. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. Default value is implementation-defined.
mcapi_endp_attr_priority_t	This attribute defines the endpoint priority, applied to a channel at the time the channel is connected. This is a channel compatibility attribute, meaning that a channel connection requires that this attribute value is the same for both channel endpoints. A lower number means higher priority. A value of MCAPI_MAX_PRORITY (0) denotes the highest priority.
mcapi_endp_attr_num_send_buffers_t	This attribute contains the number of send buffers at the current endpoint priority level. Default value is implementation-defined.
mcapi_endp_attr_num_recv_buffers_t	This attribute contains the number of receive buffers available. This can for example be used for throttling. Implementation-defined default value
mcapi_endp_attr_status_t	This attribute contains endpoint status flags. Flags are used to query the status of an endpoint, e.g. if it is connected and if so what type of channel, direction, etc. See Note A and STANDARD STATUS FLAGS, below this table.
mcapi_endp_attr_timeout_t	This attribute contains the timeout value for blocking send and receive functions. a value of MCAPI_TIMEOUT_IMMEDIATE (or 0) means that the function will return "immediately", with success or failure. MCAPI_TIMEOUT_INFINITE means that the function will block until it completes with success or failure. Default = MCAPI_TIMEOUT_INFINITE.

Note A: The lower 16 bits are defined in mcapi.h whereas the upper 16 bits are reserved for implementation specific purposes and if used must be defined in implementation_spec.h. It is therefore recommended that the upper 16 bits be masked off at the application level.

```
          0x00000000
                ----  mcapi.h
            ----            implementation_spec.h

Default = 0x00000000
```

STANDARD STATUS FLAGS

MCAPI_ENDP_ATTR_STATUS_CONNECTED	The endpoint is one end of a connected channel.
MCAPI_ENDP_ATTR_STATUS_OPEN	A channels is open on this endpoint.
MCAPI_ENDP_ATTR_STATUS_OPEN_PENDING	A channel open is pending.
MCAPI_ENDP_ATTR_STATUS_CLOSE_PENDING	A channel close is pending.
MCAPI_ENDP_ATTR_STATUS_PKTCHAN	Packet channel.
MCAPI_ENDP_ATTR_STATUS_SCLCHAN	Scalar channel.
MCAPI_ENDP_ATTR_STATUS_SEND	Send side.
MCAPI_ENDP_ATTR_STATUS_RECEIVE	Receive side.

RETURN VALUE

On success, *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not an endpoint descriptor.
MCAPI_ERR_ATTR_NUM	Unknown attribute number.
MCAPI_ERR_ATTR_VALUE	Incorrect attribute value.
MCAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MCAPI_ERR_ATTR_NOTSUPPORTED	Attribute not supported by the implementation.
MCAPI_ERR_ATTR_READONLY	Attribute cannot be modified.
MCAPI_ERR_CHAN_CONNECTED	Attribute changes not allowed on connected endpoints.
MCAPI_ERR_ENDP_REMOTE	Attributes can only be set on local endpoints.
MCAPI_ERR_PARAMETER	Incorrect attribute parameter (NULL pointer).
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_endpoint_set_attribute()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.4 Messages

MCAPI Messages facilitate connectionless transfer of data buffers. The messaging API provides a "user-specified buffer" communications interface – the programmer specifies a buffer of data to be sent on the send side, and the user specifies an empty buffer to be filled with incoming data on the receive side. The implementation must be able to transfer messages to and from any buffer the programmer specifies, although the implementation may use extra buffering internally to queue up data between the sender and receiver.

3.4.1 MCAPI_MSG_SEND_I

NAME

mcapi_msg_send_i

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends a (connectionless) message from a send endpoint to a receive endpoint. It is a non-blocking function, and returns immediately. `send_endpoint`, is a local endpoint identifying the send endpoint, `receive_endpoint` identifies a receive endpoint. `buffer` is the application provided buffer, `buffer_size` is the buffer size in bytes. `priority` determines the message priority with a value of 0 being the highest priority and `request` is the identifier used to determine if the send operation has completed on the sending endpoint and the buffer can be reused by the application. Furthermore, this method will abandon the send and return `MCAPI_ERR_MEM_LIMIT` if sufficient memory space is not available. This function cannot be used to send a message to a connected endpoint. Implementations may chose to prevent messages from being sent to connected endpoint or to leave it up to the application to manage this. Functionality for this may be added in a future version of MCAPI in it is therefore recommended that implementations preventing messages from being sent to connected endpoint use `MCAPI_ERR_GENERAL` to report an error. The behavior should be documented.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	One or both endpoints are invalid.
MCAPI_ERR_MSG_SIZE	The message size exceeds the maximum size allowed by the MCAPI implementation.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_MEM_LIMIT	No memory available.
MCAPI_ERR_PRIORITY	Incorrect priority level.
MCAPI_ERR_TRANSMISSION	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
MCAPI_ERR_PARAMETER	Incorrect request or buffer (applies if buffer = NULL and buffer_size > 0) parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test(), mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.4.2 MCAPI_MSG_SEND

NAME

mcapi_msg_send

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends a (connectionless) message from a send endpoint to a receive endpoint. It is a blocking function, and returns once the buffer can be reused by the application. `send_endpoint` is a local endpoint identifying the send endpoint and `receive_endpoint` identifies a receive endpoint. buffer is the application provided buffer and `buffer_size` is the buffer size in bytes. `priority` determines the message priority with a value of 0 being the highest priority. This method will block if there is insufficient memory space available. When sufficient space becomes available, the function will complete. This function cannot be used allowed to send a message to a connected endpoint. Implementations may chose to prevent messages from being sent to connected endpoint or to leave it up to the application to manage this. Functionality for this may be added in a future version of MCAPI in it is therefore recommended that implementations preventing messages from being sent to connected endpoint use `MCAPI_ERR_GENERAL` to report an error. The behavior should be documented.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below. Success means that the entire buffer has been sent.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_ENDP_INVALID`	One or both endpoints are invalid.
`MCAPI_ERR_MSG_SIZE`	The message size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ERR_PRIORITY`	Incorrect priority level.
`MCAPI_ERR_TRANSMISSION`	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
`MCAPI_ERR_PARAMETER`	Incorrect buffer (applies if buffer = NULL and buffer_size > 0) parameter.
`MCAPI_TIMEOUT`	The operation timed out. Implementations can optionally support timeout for this function. The timeout value is set with endpoint attributes.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.4.3 MCAPI_MSG_RECV_I

NAME

mcapi_msg_recv_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_msg_recv_i(
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT void* buffer,
   MCAPI_IN size_t buffer_size,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a (connectionless) message from a receive endpoint. It is a non-blocking function, and returns immediately. `receive_endpoint` is a local endpoint identifying the receive endpoint. buffer is the application provided buffer, and `buffer_size` is the buffer size in bytes. `request` is the identifier used to determine if the receive operation has completed (all the data is in the buffer). Furthermore, this method will abandon the receive and return `MCAPI_ERR_MEM_LIMIT` if the system cannot either wait for sufficient memory to become available or allocate enough memory.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not a valid local endpoint descriptor.
MCAPI_ERR_MSG_TRUNCATED	The message size exceeds the buffer_size.
MCAPI_ERR_TRANSMISSION	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_MEM_LIMIT	No memory available.
MCAPI_ERR_PARAMETER	Incorrect buffer and/or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.4.4 MCAPI_MSG_RECV

NAME

mcapi_msg_recv

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Receives a (connectionless) message from a receive endpoint. It is a blocking function, and returns once a message is available and the received data filled into the buffer. `receive_endpoint` is a local endpoint identifying the receive endpoint. `buffer` is the application provided buffer, and `buffer_size` is the buffer size in bytes. The `received_size` parameter is filled with the actual size of the received message. This method will block if there is insufficient memory space available. When sufficient space becomes available, the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not a valid local endpoint descriptor.
MCAPI_ERR_MSG_TRUNCATED	The message size exceeds the `buffer_size`.
MCAPI_ERR_TRANSMISSION	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
MCAPI_ERR_PARAMETER	Incorrect buffer and/or `received_size` parameter.
MCAPI_TIMEOUT	The operation timed out. Implementations can optionally support timeout for this function. The timeout value is set with endpoint attributes.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.4.5 MCAPI_MSG_AVAILABLE

NAME

`mcapi_msg_available`

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_msg_available(
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if messages are available on a local receive endpoint. The function returns in a timely fashion. The number of "available" incoming messages is defined as the number of `mcapi_msg_recv()` operations that are guaranteed to not block waiting for incoming data. receive_endpoint is a local identifier for the receive endpoint. The call only checks the availability of messages and does not dequeue them. `mcapi_msg_available()` can only be used to check availability on endpoints on the node local to the caller.

RETURN VALUE

On success, the number of available messages is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` (or 0) is returned and `*mcapi_status` is set to the appropriate error defined below. `MCAPI_NULL` (or 0) could be a valid number of available messages, so status has to be checked to ensure correctness.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_ENDP_INVALID`	Argument is not a valid local endpoint descriptor.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.5 Packet Channels

MCAPI packet channels transfer data packets between a pair of connected endpoints. A connection between two endpoints is established via a two-phase process. First, some node in the system calls `mcapi_pktchan_connect_i()` to define a connection between two endpoints. This function returns immediately. In the second phase, both sender and receiver open their end of the channel by invoking `mcapi_pktchan_send_open_i()` and `mcapi_pktchan_recv_open_i()`, respectively. The connection is synchronized when both the sender and receiver open functions have completed. In order to avoid deadlock situations, the open functions are non-blocking.

This two-phased binding approach has several important benefits. The “connect” call can be made by any node in the system, which allows the programmer to define the entire channel topology in a single piece of code. This code could even be auto-generated by some static connection tool. This makes it easy to change the channel topology without having to modify multiple source files. This approach also allows the sender and receiver to do their work without any knowledge of what remote nodes they are connected to. This allows for better modularity and application scaling.

Packet channels provide a “system-specified buffer” interface. The programmer specifies the address of a buffer of data to be sent on the send side, but the receiver's `recv` function returns a buffer of data at an address chosen by the system. This is different from the “user-specified buffer” interface use by MCAPI messaging – with messages the programmer chooses the buffer in which data is received, and with packet channels the system chooses the buffer.

It is not allowed to send messages to connected endpoints.

3.5.1 MCAPI_PKTCHAN _CONNECT_I

NAME

`mcapi_pktchan_connect_i`

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_connect_i(
   MCAPI_IN mcapi_endpoint_t send_endpoint,
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Connects a pair of endpoints into a unidirectional FIFO channel. The connect operation can be performed by the sender, the receiver, or by a third party. The connect can happen at the start of the program, or dynamically at run time.

Connect is a non-blocking function. Synchronization to ensure the channel has been created is provided by the open call discussed later.

Attempts to make multiple connections to a single endpoint will be detected as errors. The type of channel connected to an endpoint must match the type of open call invoked by that endpoint; the open function will return an error if the opened channel type does not match the connected channel type, or direction.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (whether attributes are compatible or not is implementation-defined). It is also an error to attempt to change the attributes of endpoints that are connected. It is an error to connect an endpoint with itself. A previously connected endpoint can't be connected until the previous channel is disconnected (implies both sides closed).

RETURN VALUE

On success `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not a valid endpoint descriptor, or both endpoints are the same.
MCAPI_ERR_CHAN_CONNECTED	A channel connection has already been established for one or both of the specified endpoints.
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_ATTR_INCOMPATIBLE	Connection of endpoints with incompatible attributes not allowed.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.2 MCAPI_PKTCHAN_RECV_OPEN_I

NAME

mcapi_pktchan_recv_open_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv_open_i(
   MCAPI_OUT mcapi_pktchan_recv_hndl_t* receive_handle,
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the receive end of a packet channel. The corresponding calls are required on both sides for synchronization to ensure that the channel has been created. It is a non-blocking function, and the receive_handle is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. receive_endpoint is the local endpoint associated with the channel. The open call returns a typed, local handle for the connected channel that is used for channel receive operations. An endpoint with a previously open channel can't be opened until the previous channel is disconnected (implies both sides closed).

RETURN VALUE

On success, meaning that both sides of the channel are successfully open, *mcapi_status is set to MCAPI_SUCCESS if completed and MCAPI_PENDING if not yet completed. On error *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not a valid local endpoint descriptor.
MCAPI_ERR_CHAN_TYPE	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to open a receive handle on an endpoint that was connected as a sender.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_CHAN_OPENPENDING	An open request is pending.
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_CHAN_OPEN	The channel is already open.
MCAPI_ERR_ENDP_DELETED	The endpoint has been deleted.
MCAPI_ERR_PARAMETER	Incorrect handle or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the mcapi_test() , mcapi_wait() and mcapi_wait_any() functions to query the status and mcapi_cancel() function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.3 MCAPI_PKTCHAN_SEND_OPEN_I

NAME

`mcapi_pktchan_send_open_i`

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_send_open_i(
   MCAPI_OUT mcapi_pktchan_send_hndl_t* send_handle,
   MCAPI_IN mcapi_endpoint_t send_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the send end of a packet channel. The corresponding calls are required on both sides for synchronization to ensure that the channel has been created. It is a non-blocking function, and the `send_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `send_endpoint` is the local endpoint associated with the channel. The open call returns a typed, local handle for the connected endpoint that is used by channel send operations. An endpoint with a previously open channel can't be opened until the previous channel is disconnected (implies both sides closed).

RETURN VALUE

On success, meaning that both sides of the channel are successfully open, *mcapi_status is set to MCAPI_SUCCESS if completed and MCAPI_PENDING if not yet completed. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_ENDP_INVALID`	Argument is not a valid local endpoint descriptor.
`MCAPI_ERR_CHAN_TYPE`	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
`MCAPI_ERR_CHAN_DIRECTION`	Attempt to open a send handle on a port that was connected as a receiver.
`MCAPI_ERR_REQUEST_LIMIT`	No more request handles available.
`MCAPI_ERR_CHAN_OPENPENDING`	An open request is pending.
`MCAPI_ERR_CHAN_CLOSEPENDING`	A close request is pending.
`MCAPI_ERR_CHAN_OPEN`	The channel is already open.
`MCAPI_ERR_ENDP_DELETED`	The endpoint has been deleted.
`MCAPI_ERR_PARAMETER`	Incorrect handle or request parameter.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test() , mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.4 MCAPI_PKTCHAN_SEND_I

NAME

`mcapi_pktchan_send_i`

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends a packet on a connected channel. It is a non-blocking function, and returns immediately. buffer is the application provided buffer and size is the buffer size. request is the identifier used to determine if the send operation has completed on the sending endpoint and the buffer can be reused. While this method returns immediately, data transfer will not complete until the packet has been transmitted. The definition of transmission in this context is implementation-defined and may include blocking until the send buffer is available for reuse. Alternatively the buffer's availability for reuse can be tested with the `mcapi_pktchan_release_test()` function. The behavior must be documented by the implementation. Furthermore, this method will abandon the send and return `MCAPI_ERR_MEM_LIMIT` if the system cannot either wait for sufficient memory to become available or allocate enough memory.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_CHAN_INVALID`	Argument is not a valid channel handle.
`MCAPI_ERR_PKT_SIZE`	The packet size exceeds the maximum size allowed by the MCAPI implementation.
`MCAPI_ERR_REQUEST_LIMIT`	No more request handles available.
`MCAPI_ERR_MEM_LIMIT`	No memory available.
`MCAPI_ERR_TRANSMISSION`	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
`MCAPI_ERR_PARAMETER`	Incorrect request or buffer (applies if buffer = 0 and buffer_size > 0) parameter.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.5 MCAPI_PKTCHAN_SEND

NAME

mcapi_pktchan_send

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends a packet on a connected channel. This method will block until the packet has been transmitted. The definition of transmission in this context is implementation-defined and may include blocking until the send buffer is available for reuse. Alternatively the buffer's availability for reuse can be tested with the `mcapi_pktchan_release_test()` function. The behavior must be documented by the implementation. `send_handle` is the local send handle which represents the send endpoint associated with the channel. `buffer` is the application provided buffer and `size` is the buffer size. Since channels behave like FIFOs, by default this method will block if the packet can't be transmitted because of lack of memory space. When sufficient space becomes available, the function will complete. By default this method will block if there is insufficient memory space available. When sufficient space becomes available, the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below. Success means that the entire buffer has been sent.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a valid channel handle.
MCAPI_ERR_PKT_SIZE	The message size exceeds the maximum size allowed by the MCAPI implementation.
MCAPI_ERR_TRANSMISSION	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
MCAPI_ERR_PARAMETER	Incorrect buffer (applies if buffer = 0 and buffer_size > 0) parameter.
MCAPI_TIMEOUT	The operation timed out. Implementations can optionally support timeout for this function. The timeout value is set with endpoint attributes.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.6 MCAPI_PKTCHAN_RECV_I

NAME

mcapi_pktchan_recv_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv_i(
   MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
   MCAPI_OUT void** buffer,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a packet on a connected channel. It is a non-blocking function, and returns immediately. `receive_handle` is the local representation of the handle used to receive packets. When the receive operation completes, the `buffer` parameter is filled with the address of a system-supplied buffer containing the received packet. After the receive request has completed and the application is finished with `buffer`, `buffer` must be returned to the system by calling `mcapi_pktchan_release()`. `request` is the identifier used to determine if the receive operation has completed and `buffer` is ready for use; the `mcapi_test()`, `mcapi_wait()` or `mcapi_wait_any()` function will return the actual size of the received packet. Furthermore, this method will abandon the receive and return `MCAPI_ERR_MEM_LIMIT` if sufficient memory space is not available.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a valid channel handle.
MCAPI_ERR_MEM_LIMIT	No memory available
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_TRANSMISSION	Transmission failure. This error code is optional, and if supported by an implementation, its functionality shall be described.
MCAPI_ERR_PARAMETER	Incorrect buffer or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.7 MCAPI_PKTCHAN_RECV

NAME

`mcapi_pktchan_recv`

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv(
   MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
   MCAPI_OUT void** buffer,
   MCAPI_OUT size_t* received_size,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a packet on a connected channel. It is a blocking function, and returns when the data has been written to the buffer. `receive_handle` is the local representation of the receive endpoint associated with the channel. When the receive operation completes, the buffer parameter is filled with the address of a system-supplied buffer containing the received packet. and `received_size` is filled with the size of the packet in that buffer. When the application finishes with buffer, it must return it to the system by calling `mcapi_pktchan_release()`. By default this method will block if there is insufficient memory space available. When sufficient space becomes available, the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_CHAN_INVALID`	Argument is not a valid channel handle.
`MCAPI_ERR_TRANSMISSION`	Transmission failure. This error code is optional, and if supported by an implementation, it's functionality shall be described.
`MCAPI_ERR_PARAMETER`	Incorrect buffer or `received_size` parameter.
`MCAPI_TIMEOUT`	The operation timed out. Implementations can optionally support timeout for this function. The timeout value is set with endpoint attributes.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.5.8 MCAPI_PKTCHAN_AVAILABLE

NAME

mcapi_pktchan_available

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_pktchan_available(
   MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if packets are available on a receive endpoint. This function returns in a timely fashion. The number of available packets is defined as the number of receive operations that could be performed without blocking to wait for incoming data. `receive_handle` is the local handle for the packet channel. The call only checks the availability of packets and does not dequeue them.

RETURN VALUE

On success, the number of available packets are returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` (or 0) is returned and `*mcapi_status` is set to the appropriate error defined below. `MCAPI_NULL` (or 0) could be a valid number of available packets, so status has to be checked to ensure correctness.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_CHAN_INVALID`	Argument is not a channel handle.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.9 MCAPI_PKTCHAN_RELEASE

NAME

mcapi_pktchan_release

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_release(
   MCAPI_IN void* buffer,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

When a user is finished with a packet buffer obtained from mcapi_pktchan_recv_i() or mcapi_pktchan_recv(), they must invoke this function to return the buffer to the system. Buffers can be released in any order. This function is guaranteed to return in a timely fashion. Buffers can be released by the receiving node only.

RETURN VALUE

On success *mcapi_status is set to MCAPI_SUCCESS. On error, *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_BUF_INVALID	Argument is not a valid buffer descriptor (including NULL).
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_pktchan_recv(), mcapi_pktchan_recv_i()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.10 MCAPI_PKTCHAN_RELEASE_TEST

NAME

mcapi_pktchan_release_test

SYNOPSIS

```
#include <mcapi.h>

mcapi_boolean_t mcapi_pktchan_release_test(
   MCAPI_IN void* buffer,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if a packet buffer obtained from a mcapi_pktchan_recv() has been released with mcapi_pktchan_release by the receiver. This function is blocking and returns in a timely fashion.

RETURN VALUE

On success, MCAPI_TRUE is returned and *mcapi_status is set to MCAPI_SUCCESS. If the operation has not completed MCAPI_FALSE is returned and *mcapi_status is set to MCAPI_PENDING. On error MCAPI_FALSE is returned and *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_PARAMETER	Incorrect buffer parameter (NULL).
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_pktchan_release()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.11 MCAPI_PKTCHAN_RECV_CLOSE_I

NAME

mcapi_pktchan_recv_close_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_recv_close_i(
   MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the receive side of a channel. The sender makes the send-side call and the receiver makes the receive-side call. The corresponding calls are required on both sides to ensure that the channel has been properly closed. It is a non-blocking function, and returns immediately. `receive_handle` is the local representation of the handle used to receive packets. All pending packets are discarded, and any attempt to send more packets will give an error. A packet channel is disconnected when, both sides have issued close calls and the last (second) close operation is performed.

RETURN VALUE

On success, meaning that both sides of the channel are successfully closed, `*mcapi_status` is `set to MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a valid channel handle.
MCAPI_ERR_CHAN_TYPE	Attempt to close a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to close a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_NOTOPEN	The channel is not open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test() , mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.5.12 MCAPI_PKTCHAN_SEND_CLOSE_I

NAME

mcapi_pktchan_send_close_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_pktchan_send_close_i(
   MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the send side of a channel. The sender makes the send-side call and the receiver makes the receive-side call. The corresponding calls are required on both sides to ensure that the channel has been properly closed. It is a non-blocking function, and returns immediately. `send_handle` is the local representation of the handle used to send packets. Pending packets at the receiver are not discarded. A packet channel is disconnected when, both sides have issued close calls and the last (second) close operation is performed.

RETURN VALUE

On success, meaning that both sides of the channel are successfully closed, `*mcapi_status` is set to `MCAPI_SUCCESS` if complete`d and MCAPI_PENDING` if not yet completed. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_CHAN_TYPE	Attempt to close a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to close a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_NOTOPEN	The channel is not open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test() , mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6 Scalar Channels

MCAPI scalar channels are used to transfer 8-bit, 16-bit, 32-bit and 64-bit scalars on a connected channel. The connection process for scalar channels uses the same two-phase mechanism as packet channels. See the packet channel section for a detailed description of the connection process.

The MCAPI scalar channels API provides only blocking send and receive methods. Scalar channels are intended to provide a very low overhead interface for moving a stream of values. In fact, some embedded systems may be able to implement a scalar channel as a hardware FIFO. The sort of streaming algorithms that take advantage of scalar channels should not require a non-blocking send or receive method; each process should simply receive a value to work on, do its work, send the result out on a channel, and repeat. Applications that require non-blocking semantics should use packet channels instead of scalar channels. The scalar functions only support communication of same size scalars on both sides, i.e. an 8-bit send must be read by an 8-bit receive, etc.

3.6.1 MCAPI_SCLCHAN_CONNECT_I

NAME

`mcapi_sclchan_connect_i`

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_connect_i(
   MCAPI_IN mcapi_endpoint_t send_endpoint,
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Connects a pair of endpoints into a unidirectional FIFO channel. The connect operation can be performed by the sender, the receiver, or by a third party. The connect can happen once at the start of the program or dynamically at run time.

`mcapi_sclchan_connect_i()` is a non-blocking function. Synchronization to ensure the channel has been created is provided by the open call discussed later.

This function behaves like the packet channel connect call.

Attempts to make multiple connections to a single endpoint will be detected as errors. The type of channel connected to an endpoint must match the type of open call invoked by that endpoint; the open function will return an error if the opened channel type does not match the connected channel type, or direction.

It is an error to attempt a connection between endpoints whose attributes are set in an incompatible way (whether attributes are compatible or not is implementation-defined). It is also an error to attempt to change the attributes of endpoints that are connected. It is an error to connect an endpoint with itself. A previously connected endpoint can't be connected until the previous channels is disconnected (implies both sides closed).

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not a valid endpoint descriptor, or both endpoints are the same.
MCAPI_ERR_CHAN_CONNECTED	A channel connection has already been established for one or both of the specified endpoints.
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_ATTR_INCOMPATIBLE	Connection of endpoints with incompatible attributes not allowed.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status of and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.2 MCAPI_SCLCHAN_RECV_OPEN_I

NAME

mcapi_sclchan_recv_open_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_recv_open_i(
   MCAPI_OUT mcapi_sclchan_recv_hndl_t* receive_handle,
   MCAPI_IN mcapi_endpoint_t receive_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the receive end of a scalar channel. It also provides synchronization for channel creation between two endpoints. The corresponding calls are required on both sides to synchronize the endpoints. It is a non-blocking function, and the `recv_handle` is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `receive_endpoint` is the local endpoint identifier. The call returns a local handle for the connected channel. An endpoint with a previously open channel cannot be opened until the previous channel is disconnected (implies both sides closed).

RETURN VALUE

On success, meaning that both sides of the channel are successfully open, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not an endpoint descriptor. A remote endpoint is also invalid for this function.
MCAPI_ERR_CHAN_TYPE	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_OPENPENDING	An open request is pending.
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_CHAN_OPEN	The channel is already open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_ENDP_DELETED	The endpoint has been deleted.
MCAPI_ERR_PARAMETER	Incorrect handle or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test()` , `mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.3 MCAPI_SCLCHAN_SEND_OPEN_I

NAME

mcapi_sclchan_send_open_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_open_i(
   MCAPI_OUT mcapi_sclchan_send_hndl_t* send_handle,
   MCAPI_IN mcapi_endpoint_t send_endpoint,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Opens the send end of a scalar channel. It also provides synchronization for channel creation between two endpoints. The corresponding calls are required on both sides to synchronize the endpoints. It is a non-blocking function, and the send_handle is filled in upon successful completion. No specific ordering of calls between sender and receiver is required since the call is non-blocking. `send_endpoint` is the local endpoint identifier. The call returns a local handle for connected channel. An endpoint with a previously open channel can't be opened until the previous channel is disconnected (implies both sides closed).

RETURN VALUE

On success, meaning that both sides of the channel are successfully open, `*mcapi_status` is set to `MCAPI_SUCCESS` if completed and `MCAPI_PENDING` if not yet completed. On error, `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_ENDP_INVALID	Argument is not an endpoint descriptor. A remote endpoint is also invalid for this function.
MCAPI_ERR_CHAN_TYPE	Attempt to open a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to open a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_OPENPENDING	An open request is pending.
MCAPI_ERR_CHAN_CLOSEPENDING	A close request is pending.
MCAPI_ERR_CHAN_OPEN	The channel is already open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_ENDP_DELETED	The endpoint has been deleted.
MCAPI_ERR_PARAMETER	Incorrect handle or request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the `mcapi_test() , mcapi_wait()` and `mcapi_wait_any()` functions to query the status and `mcapi_cancel()` function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.4 MCAPI_SCLCHAN_SEND_UINT64

NAME

mcapi_sclchan_send_uint64

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint64(
   MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
   MCAPI_IN mcapi_uint64_t dataword,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a 64-bit scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.5 MCAPI_SCLCHAN_SEND_UINT32

NAME

mcapi_sclchan_send_uint32

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_uint32(
   MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
   MCAPI_IN mcapi_uint32_t dataword,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Sends a 32-bit scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.6 MCAPI_SCLCHAN_SEND_UINT16

NAME

mcapi_sclchan_send_uint16

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends a 16-bit scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.7 MCAPI_SCLCHAN_SEND_UINT8

NAME

mcapi_sclchan_send_uint8

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Sends an 8-bit scalar on a connected channel. It is a blocking function, and returns immediately unless the buffer is full. `send_handle` is the send endpoint identifier. `dataword` is the scalar. Since channels behave like FIFOs, this method will block if there is no free space in the channel's receive buffer. When sufficient space becomes available (due to receive calls), the function will complete.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.8 MCAPI_SCLCHAN_RECV_UINT64

NAME

mcapi_sclchan_recv_uint64

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint64_t mcapi_sclchan_recv_uint64(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a 64-bit scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint64_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.9 MCAPI_SCLCHAN_RECV_UINT32

NAME

mcapi_sclchan_recv_uint32

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint32_t mcapi_sclchan_recv_uint32(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a 32-bit scalar on a connected channel. It is a blocking function, and returns when a scalar is available. `receive_handle` is the receive endpoint identifier.

RETURN VALUE

On success, a value of type `uint32_t` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, the return value is undefined and `*mcapi_status` is set to the appropriate error defined below. Optionally, implementations may choose to always set `*mcapi_status` to `MCAPI_SUCCESS` for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.10 MCAPI_SCLCHAN_RECV_UINT16

NAME

mcapi_sclchan_recv_uint16

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint16_t mcapi_sclchan_recv_uint16(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives a 16-bit scalar on a connected channel. It is a blocking function, and returns when a scalar is available. receive_handle is the receive endpoint identifier.

RETURN VALUE

On success, a value of type uint16_t is returned and *mcapi_status is set to MCAPI_SUCCESS. On error, the return value is undefined and *mcapi_status is set to the appropriate error defined below. Optionally, implementations may choose to always set *mcapi_status to MCAPI_SUCCESS for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.11 MCAPI_SCLCHAN_RECV_UINT8

NAME

mcapi_sclchan_recv_uint8

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint8_t mcapi_sclchan_recv_uint8(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Receives an 8-bit scalar on a connected channel. It is a blocking function, and returns when a scalar is available. receive_handle is the receive endpoint identifier.

RETURN VALUE

On success, a value of type uint8_t is returned and *mcapi_status is set to MCAPI_SUCCESS. On error, the return value is undefined and *mcapi_status is set to the appropriate error defined below. Optionally, implementations may choose to always set *mcapi_status to MCAPI_SUCCESS for performance reasons.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The receive scalar size must match the send size.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.12 MCAPI_SCLCHAN_AVAILABLE

NAME

`mcapi_sclchan_available`

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_sclchan_available(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Checks if scalars are available on a receive endpoint. The function returns immediately. receive_endpoint is the receive endpoint identifier. The call only checks the availability of messages does not dequeue them.

RETURN VALUE

On success, the number of available scalars is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, `MCAPI_NULL` (or 0)is returned and `*mcapi_status` is set to the appropriate error defined below. `MCAPI_NULL` (or 0) could be a valid number of available scalars, so status has to be checked to ensure correctness.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_CHAN_INVALID`	Argument is not a channel handle.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

The status code must be checked to distinguish between no messages and an error condition.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

3.6.13 MCAPI_SCLCHAN_RECV_CLOSE_I

NAME

mcapi_sclchan_recv_close_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_recv_close_i(
   MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the receive side of a channel. The corresponding calls are required on both send and receive sides to ensure that the channel is properly closed. It is a non-blocking function, and returns immediately. receive_handle is the local representation of the handle used to receive packets. All pending scalars are discarded, and any attempt to send more scalars will give an error. A scalar channel is disconnected when, both sides have issued close calls and the last (second) close operation is performed.

RETURN VALUE

On success, meaning that both sides of the channel are successfully closed, *mcapi_status is set to MCAPI_SUCCESS if completed and MCAPI_PENDING if not yet completed. On error *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_CHAN_TYPE	Attempt to close a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to close a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_NOTOPEN	The channel is not open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the mcapi_test() , mcapi_wait() and mcapi_wait_any() functions to query the status of and mcapi_cancel() function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.6.14 MCAPI_SCLCHAN_SEND_CLOSE_I

NAME

mcapi_sclchan_send_close_i

SYNOPSIS

```
#include <mcapi.h>

void mcapi_sclchan_send_close_i(
   MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
   MCAPI_OUT mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Closes the send side of a channel. The corresponding calls are required on both send and receive sides to ensure that the channel is properly closed. It is a non-blocking function, and returns immediately. send_handle is the local representation of the handle used to send packets. Pending scalars at the receiver are not discarded. A scalar channel is disconnected when, both sides have issued close calls and the last (second) close operation is performed.

RETURN VALUE

On success, meaning that both sides of the channel are successfully closed, *mcapi_status is set to MCAPI_SUCCESS if completed and MCAPI_PENDING if not yet completed. On error *mcapi_status is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_CHAN_INVALID	Argument is not a channel handle.
MCAPI_ERR_CHAN_TYPE	Attempt to close a packet channel on an endpoint that has been connected with a different channel type.
MCAPI_ERR_CHAN_DIRECTION	Attempt to close a send handle on a port that was connected as a receiver, or vice versa.
MCAPI_ERR_CHAN_NOTOPEN	The channel is not open.
MCAPI_ERR_REQUEST_LIMIT	No more request handles available.
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

NOTE

Use the mcapi_test() , mcapi_wait() and mcapi_wait_any() functions to query the status of and mcapi_cancel() function to cancel the operation.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.7 Non-Blocking Operations

The connectionless message and packet channel API functions have both blocking and non-blocking variants. The non-blocking versions fill in a `mcapi_request_t or mca_request_t` object and return control to the user before the communication operation is completed. The `mcapi_test()`, `mcapi_wait()`,`mcapi_wait_any()` and `mcapi_cancel()` functions are used to query the status of the non-blocking operation.

3.7.1 MCAPI_TEST

NAME

`mcapi_test`

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Checks if a non-blocking operation has completed. The function returns in a timely fashion. request is the identifier for the non-blocking operation. The call only checks the completion of an operation and doesn't affect any messages/packets/scalars and does not release the request. If the specified request completes and the pending operation was a send or receive operation, the size parameter is set to the number of bytes that were either sent or received by the non-blocking transaction. The request is released with either of `mcapi_wait(), mcapi_wait_any(),` or `mcapi_cancel()`.

RETURN VALUE

On success, `MCAPI_TRUE` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. If the operation has not completed `MCAPI_FALSE` is returned and `*mcapi_status` is set to `MCAPI_PENDING`. On error `MCAPI_FALSE` is returned and `*mcapi_status` is set to the appropriate error defined below or an error code for the corresponding non-blocking routine associated with the request structure.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle. This also applies if the request has been canceled.
`MCAPI_ERR_PARAMETER`	Incorrect size parameter.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

`mcapi_endpoint_t mcapi_endpoint_getmcapi_endpoint_get_i(), mcapi_msg_send_i(), mcapi_msg_recv_i(), mcapi_pktchan_connect_i(), mcapi_pktchan_recv_open_i(), mcapi_pktchan_send_open_i(), mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(), mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(), mcapi_sclchan_connect_i(), mcapi_sclchan_recv_open_i(), mcapi_sclchan_send_open_i(), mcapi_sclchan_recv_close_i(), mcapi_sclchan_send_close_i()`

NOTE

It is important to release the request with either of `mcapi_wait(), mcapi_wait_any(),` or `mcapi_cancel()`, in order to manage system resources.

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.7.2 MCAPI_WAIT

NAME

mcapi_wait

SYNOPSIS

```
#include <mcapi.h>

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

DESCRIPTION

Wait until a non-blocking operation has completed. It is a blocking function and returns when the operation has completed, has been canceled, or a timeout has occurred. `request` is the identifier for the non-blocking operation. The call only waits for the completion of an operation (all buffers referenced in the operation have been filled or consumed and can now be safely accessed by the application) and doesn't affect any messages/packets/scalars. The size parameter is set to the number of bytes that were either sent or received by the non-blocking transaction that completed (size is irrelevant for non-blocking connect and close calls). The `mcapi_wait()` call will return if the request is cancelled, by another thread (during the waiting), by a call to `mcapi_cancel()`, and the returned `mcapi_status` will indicate that the request was cancelled. The units for timeout are implementation-defined. If a timeout occurs the returned status will indicate that the timeout occurred. A value of 0 for the timeout parameter indicates that the function will return without blocking with failure or success and a value of `MCAPI_INFINITE` for the timeout parameter indicates no timeout is requested, i.e. the function will block until it is unblocked because of failure or success. In regards to timeouts, `mcapi_wait` is non destructive, i.e. the request is not cleared in the case of a timeout and can be waited upon multiple times. Multiple threads attempting to wait on the same request will result in an error.

RETURN VALUE

On success, `MCAPI_TRUE` is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. On error, cancellation or timeout `MCAPI_FALSE` is returned and `*mcapi_status` is set to the appropriate error/status defined below for parameter errors for the `mcapi_wait()` call or an error code for the corresponding non-blocking routine associated with the request structure.

ERRORS

`MCAPI_ERR_NODE_NOTINIT`	The node is not initialized.
`MCAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle.
`MCAPI_ERR_REQUEST_CANCELLED`	The request was canceled, by another thread (during the waiting).
`MCAPI_ERR_WAIT_PENDING`	A wait is pending.
`MCAPI_TIMEOUT`	The operation timed out.
`MCAPI_ERR_PARAMETER`	Incorrect request or size parameter.
`MCAPI_ERR_GENERAL`	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

```
mcapi_endpoint_t mcapi_endpoint_getmcapi_endpoint_get_i(),
mcapi_msg_send_i(), mcapi_msg_recv_i(), mcapi_pktchan_connect_i(),
mcapi_pktchan_recv_open_i(), mcapi_pktchan_send_open_i(),
mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(),
mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(),
mcapi_sclchan_connect_i(), mcapi_sclchan_recv_open_i(),
mcapi_sclchan_send_open_i(), mcapi_sclchan_recv_close_i(),
mcapi_sclchan_send_close_i()
```

IMPLEMENTATION SPECIFIC DETAILS

```
[This section reserved for inclusion of implementation specific details]
```

3.7.3 MCAPI_WAIT_ANY

NAME

mcapi_wait_any

SYNOPSIS

```
#include <mcapi.h>

mcapi_uint_t mcapi_wait_any(
   MCAPI_IN size_t number,
   MCAPI_IN mcapi_request_t* requests,
   MCAPI_OUT size_t* size,
   MCAPI_IN mcapi_timeout_t timeout,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Wait until any non-blocking operation of a list has completed. It is a blocking function and returns the index into the requests array (starting from 0) indicating which of any outstanding operations has completed (successfully or with an error). number is the number of requests in the array. requests is the array of `mcapi_request_t` identifiers for the non-blocking operations. The call only waits for the completion of an operation and doesn't affect any messages/packets/scalars. The size parameter is set to number of bytes that were either sent or received by the non-blocking transaction that completed (size is irrelevant for non-blocking connect and close calls). The `mcapi_wait_any()` call will return if any of the requests are cancelled, by another thread (during the waiting), by a call to `mcapi_cancel()`, and the returned `mcapi_status` will indicate that the request was cancelled.The units for timeout are implementation-defined. If a timeout occurs the `mcapi_status` parameter will indicate that a timeout occurred. A value of 0 for the timeout parameter indicates that the function will return without blocking with failure or success and a value of `MCAPI_INFINITE` for the timeout parameter indicates no timeout is requested, i.e. the function will block until it is unblocked because of failure or success. In regards to timeouts `mcapi_wait_any()` is non destructive, i.e. the request is not cleared in the case of a timeout and can be waited upon multiple times. Multiple threads attempting to wait on the same request will result in an error.

RETURN VALUE

On success, the index into the requests array of the `mcapi_request_t` identifier that has completed is returned and `*mcapi_status` is set to `MCAPI_SUCCESS`. The index is aslo returned for canceled or invalid requests and the return status will indicate that a request was cancelled or was invalid. The index may also be returned for implementation specific use of `MCAPI_ERR_GENERAL` and if so shall be documented by the implementer. For other errors or timeout, `MCAPI_RETURN_VALUE_INVALID` is returned and `*mcapi_status` is set to the appropriate error/status defined below for parameter errors for the `mcapi_wait_any()` call or errors from the requesting functions, as defined in those respective functions.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_REQUEST_INVALID	Argument is not a valid request handle.
MCAPI_ERR_REQUEST_CANCELLED	One of the requests was canceled, by another thread (during the waiting).
MCAPI_ERR_WAIT_PENDING	A wait is pending.
MCAPI_TIMEOUT	The operation timed out.
MCAPI_ERR_PARAMETER	Incorrect number (if = 0), requests or size parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_endpoint_t mcapi_endpoint_getmcapi_endpoint_get_i(), mcapi_msg_send_i(), mcapi_msg_recv_i(), mcapi_pktchan_connect_i(), mcapi_pktchan_recv_open_i(), mcapi_pktchan_send_open_i(), mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(), mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(), mcapi_sclchan_connect_i(), mcapi_sclchan_recv_open_i(), mcapi_sclchan_send_open_i(), mcapi_sclchan_recv_close_i(), mcapi_sclchan_send_close_i()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.7.4 MCAPI_CANCEL

NAME

mcapi_cancel

SYNOPSIS

```
#include <mcapi.h>

void mcapi_cancel(
   MCAPI_IN mcapi_request_t* request,
   MCAPI_OUT mcapi_status_t* mcapi_status
);
```

DESCRIPTION

Cancels an outstanding non-blocking operation. It is a blocking function and returns when the operation has been canceled. request is the identifier for the non-blocking operation. Any pending calls to `mcapi_wait()` or `mcapi_wait_any()` for this request will also be cancelled. The returned status of a canceled `mcapi_wait()` or `mcapi_wait_any()` call will indicate that the request was cancelled.

RETURN VALUE

On success, `*mcapi_status` is set to `MCAPI_SUCCESS`. On error `*mcapi_status` is set to the appropriate error defined below.

ERRORS

MCAPI_ERR_NODE_NOTINIT	The node is not initialized.
MCAPI_ERR_REQUEST_INVALID	Argument is not a valid request handle (the operation and `mcapi_wait()` or `mcapi_wait_any()` may have completed).
MCAPI_ERR_PARAMETER	Incorrect request parameter.
MCAPI_ERR_GENERAL	Implementation specific error not covered by other status codes. Specifics must be documented.

SEE ALSO

mcapi_endpoint_t mcapi_endpoint_getmcapi_endpoint_get_i(), mcapi_msg_send_i(), mcapi_msg_recv_i(), mcapi_pktchan_connect_i(), mcapi_pktchan_recv_open_i(), mcapi_pktchan_send_open_i(), mcapi_pktchan_send_i(), mcapi_pktchan_recv_i(), mcapi_pktchan_recv_close_i(), mcapi_pktchan_send_close_i(), mcapi_sclchan_connect_i(), mcapi_sclchan_recv_open_i(), mcapi_sclchan_send_open_i(), mcapi_sclchan_recv_close_i(), mcapi_sclchan_send_close_i()

IMPLEMENTATION SPECIFIC DETAILS

[This section reserved for inclusion of implementation specific details]

3.8 Support Functions

Support functions are included to simplify the use of MCAPI.

3.8.1 MCAPI_DISPLAY_STATUS

NAME

`mcapi_display_status`

SYNOPSIS

```
#include <mcapi.h>

char* mcapi_display_status(
   MCAPI_IN mcapi_status_t mcapi_status,
   MCAPI_OUT char* status message,
   MCAPI_IN size_t size
);
```

DESCRIPTION

Display an MCAPI status message in text format. It is a blocking function and returns in a timely fashion. The `status_message` parameter should point to an application provided `char` buffer of size `size` (recommended value for `size` = `MCAPI_MAX_STATUS_MSG_LEN`). `mcapi_status` is the status code. The status message string is `NULL` terminated.

RETURN VALUE

The status code in text format is returned. If the status code is invalid a `NULL` string is returned.

ERRORS

None.

IMPLEMENTATION SPECIFIC DETAILS

`[This section reserved for inclusion of implementation specific details]`

4. FAQ

Q: Is a reference implementation available? What is the intended purpose of the reference implementation?

A: An example implementation is available for download at the Multicore Association website. The example implementation models the functionality of the specification and does not intend to be a high-performance implementation.

Q: Does MCAPI provide binary interoperability?

A: MCAPI provides a set of communications primitives that supports source-code portability. Interoperability is not addressed at this stage and is left to the respective MCAPI implementations. Implementers should, however, be encouraged to ensure implementation interoperability, as much as possible. We may consider an interoperable messaging protocol in future MCAPI versions.

Q: What is the rationale behind MCAPI's use of unidirectional channels as opposed to bidirectional channels?

A: MCAPI is meant to serve as an extremely lightweight and thin communications API for use by applications and by other messaging systems (such as light-weight sockets) layered on top of it. MCAPI is expected to run on both multicore processors running real operating systems and running extremely thin run-time environments in bare-metal mode. MCAPI is also meant largely for on-chip multicore environments in which communication is reliable.

MCAPI chose to go with unidirectional channels because they use fewer resources on both the send and receive sides. A bidirectional channel would need to consume twice the resources on both the send and receive sides (for example, hardware queues).

Many of our use cases require only unidirectional channels. Although unidirectional channels could be implemented using a bidirectional channel, it would waste half the resources. Conversely, bidirectional channels can be implemented on top of unidirectional channels when desired. Locking mechanisms on the send and receive sides provided by the underlying system on top of which MCAPI is built (or provided by a layer such as MRAPI) can be used to achieve atomicity between sends and receives on each side.

Bidirectional channels are particularly effective in error-prone environments. MCAPI's target is multicore processors, where communication is expected to be reliable.

Q: How does MCAPI compare to other existing multicore communication protocols?

A: MCAPI is a low-level communication protocol specifically targeting statically configured heterogeneous multicore processors. The static property allows implementations to have lower overhead than other communication protocols such a sockets or MPI.

Q: Regarding message sends, when can the sender reuse the buffer?

A: With blocking calls, the send can reuse the buffer after the call has returned. With non-blocking calls, it is the responsibility of the sender to verify that the buffer can be reused.

Q: Can MCAPI support a software-controlled implementation of global power management? If so, how?

A: MCAPI provides communication functionality between different cores in a multicore multiprocessor and can function at user level and system level. If the global power management scheme requires data to be passed between different cores, an MCAPI implementation could be used to provide the communication functionality.

Q: Will MCAPI require modification of existing operating systems?

A: MCAPI is a communication API and does not define requirements for implementation. It is possible to implement MCAPI on top of a commercial off-the-shelf OS with no OS modifications, however an efficient implementation of MCAPI may require changes to the operating system.

Q: Can MCAPI be implemented in hardware?

A: The goal of MCAPI is to make possible efficient implementation in hardware.

Q: If MCAPI is oriented towards static connection topologies, why support methods like `mcapi_endpoint_delete()`?

A: The ability to release any resources allocated by an endpoint may be particularly important in hardware implementations in which resources are strictly limited. Supporting the deletion of endpoints allows the application programmer to dispose of endpoints that are only necessary for a portion of the application's life.

Q: What facilities are provided for debugging MCAPI programs?

A: The MCAPI API is designed to be implemented as a C library, using standard tools and enabling the use of standard debuggers and profilers. The MCAPI error codes should also help give some visibility into what sort of program errors have occurred. The MCAPI channel-connection mechanism is designed to work with static layout tools, so that a channel topology can be defined and debugged outside of the program itself. We also expect that some implementations will provide tools for dynamically collecting information like which channels have been connected and how many packets have been transferred.

Q: Why doesn't the MCAPI API support one-to-multiple, multiple-to-one. and barrier-type communications?

A: To keep the MCAPI API simple and allow for efficient implementations, more complex functions that can be layered on top of MCAPI are not part of the MCAPI specification.

Q: Why doesn't MCAPI provide name service functionality?

A: The MCAPI API is meant for multicore chips or multi-chip boards, which are static environments in the sense that the number of cores is always the same. A naming service is therefore not needed and would add unnecessary complexity. The initialization functionality added in version 2.000 provides for basic topology discovery. A naming service could be implemented on top of MCAPI.

Q: I'd like to have a name service for looking up MCAPI endpoints. How can I achieve this functionality?

A: The MCAPI standard defines a communication layer that allows easy creation of a name server. The application can choose a statically determined (domain, node, port) tuple in which the name service will run. For example, domain, node 0, port 1 might be a good choice. The name server process can be started by calling `mcapi_initialize(0,0, …)` to bind itself to domain 0, port 0 and then calling `mcapi_endpoint_create(1, &status)`. The name server can then use message send and receive operations to implement its service. Clients can discover the name server by invoking `mcapi_endpoint_get(0, 0, 1, &status)` to get the its `endpoint_t`. Given the `endpoint_t`, each client can send and receive messages as required by your name service protocol.

Q: What happens if my cores have different byte order (endian-ness)?

A: The MCAPI API specification does not state how to implement an MCAPI API. The specific implementation would have to address byte order.

Q: How do I use my MCAPI calls so that the source code is portable between implementations with different node-numbering policies?

A: To make code portable between implementations that allow only one node number (and `mcapi_initialize` call) per process, and those that allow one node number (and `mcapi_initialize` call) per thread, the code should only make one `mcapi_initialize` call per process. If there are multiple threads within the process, they should use different port numbers (endpoints) for communication.

Similarly, to make code portable between implementations that allow only one node number (and `mcapi_ initialize` call) per processor, and those that allow one node number (and `mcapi_ initialize` call) per task, the code should only make one `mcapi_ initialize` call per processor. If there are multiple tasks running on the processor, they should use different port numbers (endpoints) for communication.

Q: Does the MCAPI specification provide for multiple readers or multiple writers on an endpoint?

A: No, the specification does not provide for this. However, it does not prevent it either. An MCAPI implementation can provide a mechanism whereby multiple endpoints resolve to essentially the same destination. For example, if an SoC contained a hardware device which managed multiple hardware queues used for moving messages around the SoC, an MCAPI implementation could allow multiple writers to a single hardware queue by simply mapping a set of distinct send endpoints to that single hardware queue. The implementation might choose to do this by assigning a range of `port_ids` to map to the single hardware device.

For example, assume the MCAPI implementation identifies the hardware device that provided the hardware managed queues as `node_id` 10, and the range of ports 0..16 is assigned to map to

hardware queue zero within that device. Then a send to endpoint <0,10,0> and a send to endpoint <0,10,1> would both map to a send to queue zero in the hardware device. Different nodes within the system wishing to send to the queue would have to allocate a unique send endpoint, but the sends would all end up in queue zero in the hardware accelerator. Similarly, an MCAPI implementation could map multiple receive endpoints to a single hardware queue. In this example, any reads from multiple endpoints mapped to the single hardware queue would be serviced on a first-come, first-served basis. This would be suitable for a load balancing application, but would not be sufficient to serve as a multicast operation. Support for multicast operations will be considered in future MCAPI specification releases.

Q: How can I implement callback capability with MCAPI?

A: Although MCAPI does not directly provide a callback capability, MCAPI is designed to have many kinds of application services layered on top of it, and callbacks can be done this way. This approach would require you to use the MCAPI non-blocking send or receive functions, and to write a new function which would take the `mcapi_request_t` returned from calling these MCAPI functions along with a pointer to your callback function as parameters. This new function could use `mcapi_test`, `mcapi_wait`, or `mcapi_wait_any` to determine when the outstanding MCAPI request has completed and then call your callback function. This might be implemented by having a separate thread monitoring a list of outstanding requests and providing callbacks on completion. If you cannot create multithreaded applications your callback solution may be more difficult to implement.

5. Use Cases

5.1 Example Use of Static Naming for Initialization

MCAPI's static tuple-based naming mechanism makes it straightforward to implement a simple initialization scheme, including third-party set up of static connections. This example describes how a packet channel can be created and used using the static tuple-based naming scheme.

```
#define SENDER_DOMAIN 0
#define SENDER_NODE 0
#define SEND_PORT_ID 17

#define RECEIVER_DOMAIN 0
#define RECEIVER_NODE 1
#define RECV_PORT_ID 37
```

Sender Process:

```
mcapi_endpoint_t send_endpoint = mcapi_endpoint_create(SEND_PORT_ID,
&status);
```

Receiver Process:

```
mcapi_endpoint_t receive_endpoint = mcapi_endpoint_create(RECV_PORT_ID,
&status);
```

The connection can now be established by the receiver, the sender, or a third party process. We will use the example of a third party process.

Third party process:

```
mcapi_endpoint_t send_endpoint = mcapi_endpoint_get(SENDER_DOMAIN,
SENDER_NODE, SEND_PORT_ID, &status);
mcapi_endpoint_t receive_endpoint = mcapi_endpoint_get(RECEIVER_DOMAIN,
RECEIVER_NODE,
                                                    RECV_PORT_ID, &status);
mcapi_pktchan_connect_i(send_endpoint, receive_endpoint, …)
```

Thus, the performance impact of a global name-service lookup is entirely in the mcapi_endpoint_get() call.

5.2 Example Initialization of Dynamic Endpoints

This example describes how MCAPI performs discovery and bootstrapping when the static naming based on tuples is not being used, but instead, the dynamic endpoint scheme is used.

Whether communication is by messages or by channels, MCAPI requires a sending node to have available an endpoint on the receiving node in order to communicate. Thus, there is the following discovery and bootstrapping issue in MCAPI when static naming is not in use (and in any dynamic communication API for that matter): How to transfer the first endpoint from a receiver to a sender? This

section proposes a discovery model, which addresses the issue of how an MCAPI sender can bootstrap itself by finding the endpoint for a receiver, before any user-level communication has been established.

The basic idea is to allow the MCAPI runtime system to facilitate the creation of a root endpoint that is visible to all the nodes in the system. An endpoint can “publish” itself to the communications layer as the “root” or first-level name server of the namespace using statically known domain and node numbers. For example, the system could have exactly one statically numbered node, domain 0, node 0. All other nodes in the system would then register with node 0 via asynchronous messaging, sending it messages to let it know that they exist and what endpoints they have dynamically created. Similarly, dynamic nodes and endpoints can discover each other by sending messages to the root node. The format of the message is user-defined.

5.3 Automotive Use Case

5.3.1 Characteristics

5.3.1.1 Sensors

Tens to hundreds of sensor inputs read on a periodic basis. Each sensor is read and its data are processed by a scheduled task.

5.3.1.2 Control Task

A control task takes sensor values and computes values to apply to various actuators in the engine.

5.3.1.3 Lost Data

Lost data is not desirable, but old data quickly becomes irrelevant; the most-recent sample is most important.

5.3.1.4 Types of Tasks

Consists of both control and signal processing, especially FFT.

5.3.1.5 Load Balance

The load balance changes as engine speed increases. The frequency at which the control task must be run is determined by the RPM of the engine.

5.3.1.6 Message Size and Frequency

Messages are expected to be small and message frequency is high.

5.3.1.7 Synchronization

Synchronization between control and data tasks should be minimal to avoid negative impacts on latency of the control task. If shared memory is used there can be multiple tasks writing and one reader. Deadlock will not occur, but old data may be used if an update is not ready.

5.3.1.8 Shared Memory

Typical engine controllers incorporate on-chip flash and SRAM and can access off-chip memory as well. Shared memory regions must be in the SRAM for maximum performance. Because a small OS or no OS is involved, it is typical for logical mappings of addresses to be avoided. If an MMU is involved, it will typically be programmed for logical == physical and with few large page entries versus lots of small page entries. Maintenance of a page table and use of page-replacement algorithms should be avoided.

5.3.2 Key Functionality Requirements

5.3.2.1 Control Task

There must be a control task collecting all data and calculating updates. This task must update engine parameters continuously. Updates to engine parameters must occur when the engine crankshaft is at a particular angle, so the faster the engine is running, the more frequently this task must run.

5.3.2.2 Angle Task

There must be a data task to monitor engine RPM and schedule the control task.

5.3.2.3 Data Tasks

There must be a set of tens to hundreds of tasks to poll sensors. The task must communicate this data to the control task.

5.3.3 Context and Constraints

5.3.3.1 Operating System

Often there is no commercial operating system involved, although the notion of time-critical tasks and task scheduling must be supported by some type of executive. However, this may be changing. Possible candidates are OSEK, or other RTOS.

5.3.3.2 Polling and Interrupts

Sensor inputs may be polled and/or associated with interrupts.

5.3.3.3 Reliability

Sensors are assumed to be reliable. Interconnect is assumed to be reliable. Task completion within scheduled deadline is assumed to be reliable for the control task, and less reliable for the data tasks.

5.3.4 Metrics

5.3.4.1 Latency of Control Task

Latency of the control task depends on engine RPM. At 1800 RPM the task must complete every 33.33ms, and at 9000 RPM the task must complete every 6.667ms.

5.3.4.2 Number of Dropped Sensor Readings

Ideally zero.

5.3.4.3 Latencies of Data Tasks

Ideally the sum of the latencies plus message send/receive times should be less than the latency of the control loop, given the current engine RPM. In general, individual tasks are expected to complete in times varying from 1ms up to 1600ms, depending on the nature of the sensor and the type of processing required for its data.

5.3.4.4 Code Size

Automotive customers expect their code to fit into on-chip SRAM. The current generation of chips often has 1Mb of SRAM, with 2Mb on the near horizon.

5.3.5 Possible Factorings

- 1 general-purpose core for control, 1 general-purpose core for data
- 1 general-purpose core for control/data, dedicated SIMD core for signal processing, other special-purpose cores for remainder of data processing
- 1 core per cylinder, or 1 core per group of cylinders

5.3.6 MCAPI Requirements Implications

- Fast locks supporting multiple writers and a single reader are required. Maximum lock rate << 6ms on 800mhz core would be typical.
- Locks must work transparently whether they are unicore or multicore.
- Ability to select shared-memory region based on attribute: SRAM.
- Ability to select shared-memory region based on attribute: logical == physical.
- Ability to select shared memory region based on attribute: no MMU overhead (other than initial page-entry set up if required).

5.3.7 Mental Models

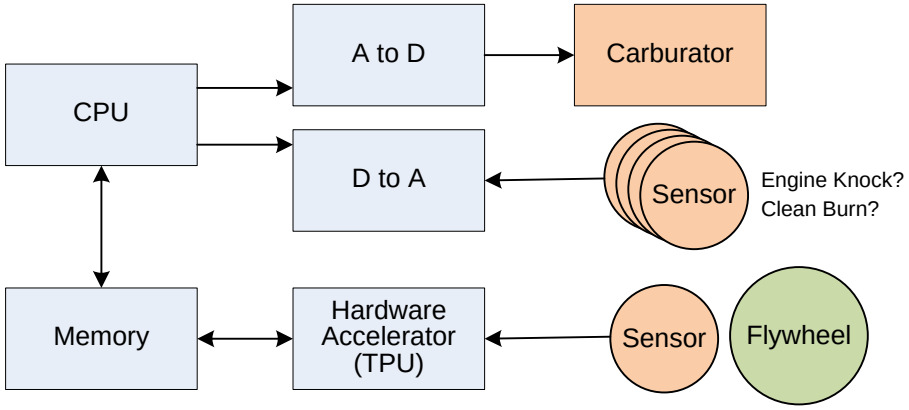


Figure 3. Example Hardware

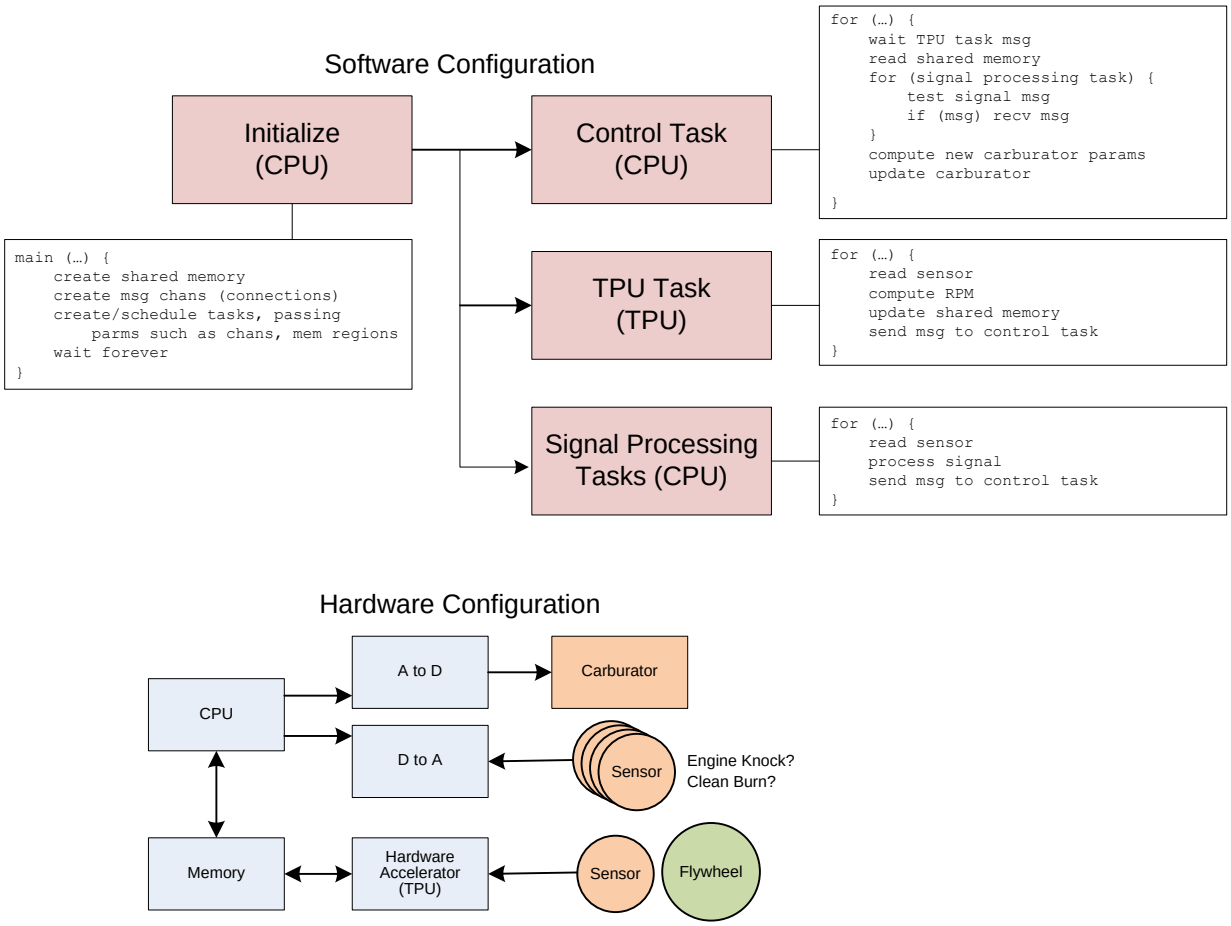


Figure 4. A Possible Mapping

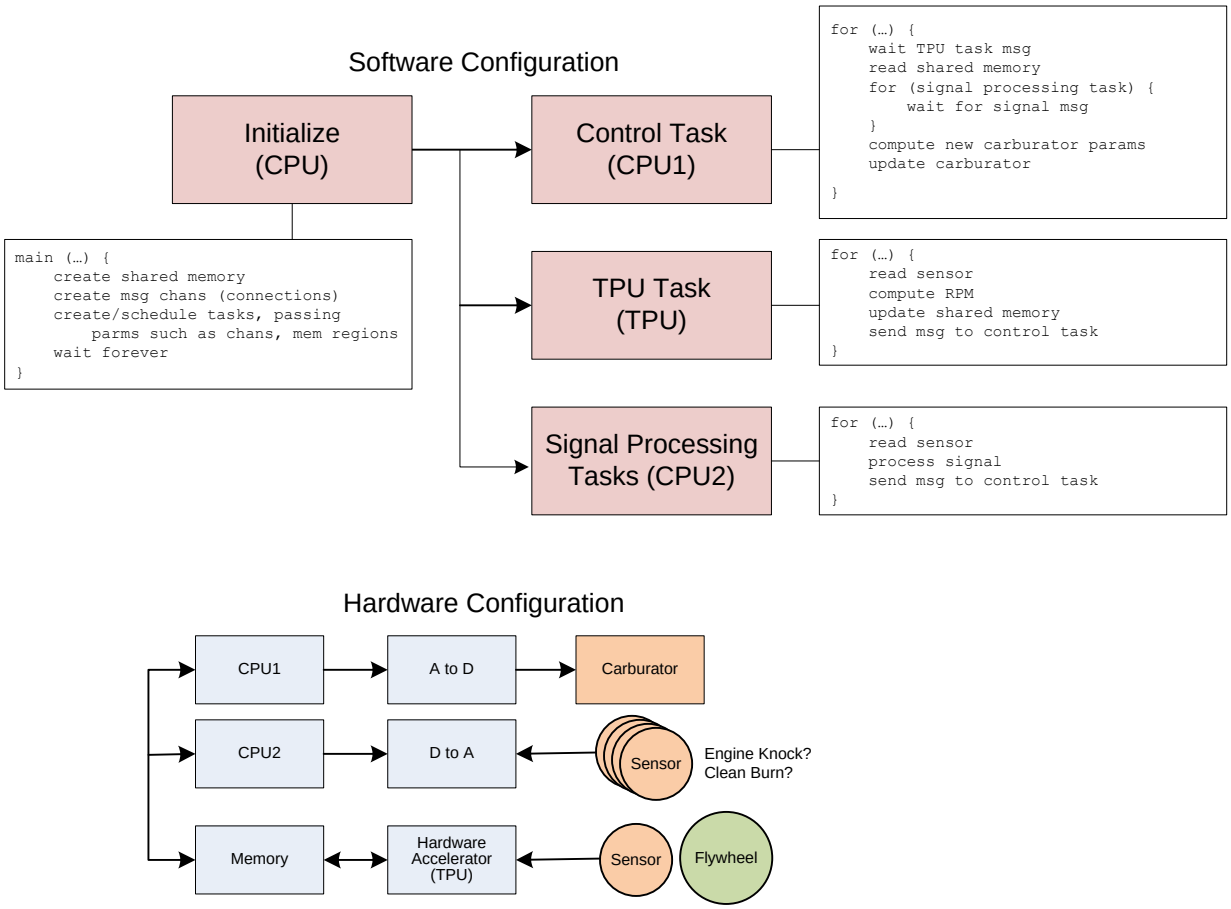


Figure 5. Alternative Hardware

5.3.8 MCAPI Pseudocode

5.3.8.1 Initial Mapping

```
/*
 * MCAPI 2.000 Automotive Use Case
 * automotive.h
 *
 */

#ifndef AUTOMOTIVE_H_
#define AUTOMOTIVE_H_

#define ENGINE_DOMAIN   0

#define CNTRL_NODE            0
#define TPU_NODE        1
#define SIG_NODE        2

#define TPU_PORT_CNTRL  0
#define CNTRL_PORT_TPU  1
#define CNTRL_PORT_SIG  2
#define SIG_PORT_CNTRL  3

#define PRIO_HIGH       0

#define SHMEM_SIZE            32

#define CHECK_STATUS(err)
#define CHECK_MEM(sMem);

typedef struct {
      int   angle;
      int   rpm;
} SIG_DATA;

#endif /* AUTOMOTIVE_H_ */

/*
 * MCAPI 2.000 Automotive Use Case
 * control_task.c
 *
 */

#include "mcapi.h"
#include "automotive.h"
extern char* shmemget(int size);
```

```
////////////////////////////////////////
// The control task
////////////////////////////////////////
void Control_Task(void) {
   char* sMem;
   mcapi_uint8_t tFlag;
   mcapi_endpoint_t tpu_endpt, tpu_remote_endpt;
   mcapi_endpoint_t sig_endpt, sig_remote_endpt;
   mcapi_endpoint_t tmp_endpt;
   mcapi_sclchan_recv_hndl_t tpu_chan;
   mcapi_pktchan_recv_hndl_t sig_chan;
   SIG_DATA sDat;
   size_t tSize;
   mcapi_request_t r1, r2;
   mcapi_status_t err;
   mcapi_priority_t priority = PRIO_HIGH;
   mcapi_node_attributes_t mcapi_node_attributes;
   mcapi_param_t mcapi_parameters;
   mcapi_info_t mcapi_info;
   mcapi_timeout_t timeout = 500;

   // init node attributes
   mcapi_node_init_attributes(&mcapi_node_attributes, &err);

   // init the system
   mcapi_initialize(ENGINE_DOMAIN, SIG_NODE, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &err);
   CHECK_STATUS(err);

   // first create two local endpoints
   tpu_endpt = mcapi_endpoint_create(CNTRL_PORT_TPU, &err);
   CHECK_STATUS(err);

   sig_endpt = mcapi_endpoint_create(CNTRL_PORT_SIG, &err);
   CHECK_STATUS(err);

   // now we get two remote endpoints
   mcapi_endpoint_get_i(ENGINE_DOMAIN, TPU_NODE, TPU_PORT_CNTRL,
            &tpu_remote_endpt, &r1, &err);
   CHECK_STATUS(err);

   sig_remote_endpt = mcapi_endpoint_get(ENGINE_DOMAIN, SIG_NODE,
            SIG_PORT_CNTRL, timeout, &err);
   CHECK_STATUS(err);

   // wait on the endpoints
   while(!((mcapi_test(&r1, &tSize, &err))) && (mcapi_test(&r2, &tSize,
            &err))) {
         // KEEP WAITING
   }

   // allocate shared memory and send the ptr to TPU task
   sMem = shmemget(32);

   tmp_endpt = mcapi_endpoint_create(MCAPI_PORT_ANY, &err);
   CHECK_STATUS(err);

   mcapi_msg_send(tmp_endpt, tpu_remote_endpt, sMem, SHMEM_SIZE, priority,
            &err);
   CHECK_STATUS(err);
```

```
   // connect the channels
   mcapi_sclchan_connect_i(tpu_endpt, tpu_remote_endpt, &r1, &err);
   CHECK_STATUS(err);

   mcapi_pktchan_connect_i(sig_endpt, sig_remote_endpt, &r2, &err);
   CHECK_STATUS(err);

   // wait on the connections
   while(!((mcapi_test(&r1, &tSize, &err))) && (mcapi_test(&r2, &tSize,
             &err))) {
       // KEEP WAITING
   }

   // now open the channels
   mcapi_sclchan_recv_open_i(&tpu_chan, tpu_endpt, &r1, &err);
   CHECK_STATUS(err);

   mcapi_pktchan_recv_open_i(&sig_chan, sig_endpt, &r2, &err);
   CHECK_STATUS(err);

   // wait on the channels
   while(!((mcapi_test(&r1, &tSize, &err))) && (mcapi_test(&r2, &tSize,
             &err))) {
       // KEEP WAITING
   }

   // now ALL of the bootstrapping is finished
   // we move to the processing phase below

   while (1) {
     // wait for TPU to indicate it has updated
     // shared memory, indicated by receipt of a flag
     tFlag = mcapi_sclchan_recv_uint8(tpu_chan, &err);
     CHECK_STATUS(err);

     // read the shared memory
     if (sMem[0] != 0) {
       // process the shared memory data
     } else {
                   // PANIC -- there was an error with the shared mem
     }

     // now get data from the signal processing task
     // would be a loop if there were multiple sig tasks
     mcapi_pktchan_recv(sig_chan, (void **) &sDat, &tSize, &err);
     CHECK_STATUS(err);

     // Compute new carb params & update carb
   }
}


/*
 * MCAPI 2.000 Automotive Use Case
 * sig_task.c
 *
 */

#include "mcapi.h"
#include "automotive.h"
```

```
//////////////////////////////////////////
// The SIG Processing Task
//////////////////////////////////////////
void SIG_task() {
   mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
   mcapi_pktchan_send_hndl_t cntrl_chan;
   mcapi_request_t r1;
   mcapi_status_t err;
   size_t size;
   mcapi_node_attributes_t mcapi_node_attributes;
   mcapi_param_t mcapi_parameters;
   mcapi_info_t mcapi_info;
   mcapi_timeout_t timeout = 500;

   // init node attributes
   mcapi_node_init_attributes(&mcapi_node_attributes, &err);

   // init the system
   mcapi_initialize(ENGINE_DOMAIN, SIG_NODE, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &err);
   CHECK_STATUS(err);

   cntrl_endpt = mcapi_endpoint_create(SIG_PORT_CNTRL, &err);
   CHECK_STATUS(err);

   mcapi_endpoint_get_i(ENGINE_DOMAIN, CNTRL_NODE, CNTRL_PORT_SIG,
             &cntrl_remote_endpt, &r1, &err);
   CHECK_STATUS(err);

   // wait on the remote endpoint
   mcapi_wait(&r1, &size, timeout, &err);
   CHECK_STATUS(err);

   // NOTE - connection handled by control task
   // open the channel
   mcapi_pktchan_send_open_i(&cntrl_chan, cntrl_endpt, &r1, &err);
   CHECK_STATUS(err);

   // wait on the open
   mcapi_wait(&r1, &size, timeout, &err);
   CHECK_STATUS(err);


   // All boostrap is finished, now begin processing
   while (1) {
        // Read sensor & process signal
        SIG_DATA sDat;  // populate this with results

      // send the data to the control process
        mcapi_pktchan_send(cntrl_chan, &sDat, sizeof(sDat), &err);
        CHECK_STATUS(err);
    }
}


/*
 * MCAPI 2.000 Automotive Use Case
 * tpu_task.c
```

```
 *
 */

#include "mcapi.h"
#include "automotive.h"

////////////////////////////////////////
// The TPU task
////////////////////////////////////////
void TPU_Task() {
   char* sMem;
   size_t msgSize;
   size_t nSize;
   mcapi_endpoint_t cntrl_endpt, cntrl_remote_endpt;
   mcapi_sclchan_send_hndl_t cntrl_chan;
   mcapi_request_t r1;
   mcapi_status_t err;
   mcapi_timeout_t timeout = 500;
   mcapi_node_attributes_t mcapi_node_attributes;
   mcapi_param_t mcapi_parameters;
   mcapi_info_t mcapi_info;

   // init node attributes
   mcapi_node_init_attributes(&mcapi_node_attributes, &err);

   // init the system
   mcapi_initialize(ENGINE_DOMAIN, SIG_NODE, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &err);
   CHECK_STATUS(err);

   cntrl_endpt = mcapi_endpoint_create(TPU_PORT_CNTRL, &err);
   CHECK_STATUS(err);

   mcapi_endpoint_get_i(ENGINE_DOMAIN, CNTRL_NODE, CNTRL_PORT_TPU,
            &cntrl_remote_endpt, &r1, &err);
   CHECK_STATUS(err);

   // wait on the remote endpoint
   mcapi_wait(&r1, &nSize, timeout, &err);
   CHECK_STATUS(err);

   // now get the shared mem ptr
   mcapi_msg_recv(cntrl_endpt, &sMem, SHMEM_SIZE, &msgSize, &err);
   CHECK_MEM(sMem);
   CHECK_STATUS(err);

   // NOTE - connection handled by control task
   // open the channel
   mcapi_sclchan_send_open_i(&cntrl_chan, cntrl_endpt, &r1, &err);
   CHECK_STATUS(err);

   // wait on the open
   mcapi_wait(&r1, NULL,timeout, &err);
   CHECK_STATUS(err);

   // ALL bootstrapping is finished, begin processing
   while (1) {
   // do something that updates shared mem
   sMem[0] = 1;

   // send a scalar flag to cntrl process
```

```
    // indicating sMem has been updated
    mcapi_sclchan_send_uint8(cntrl_chan, (mcapi_uint8_t) 1, &err);
    CHECK_STATUS(err);
    }
}
```

5.3.8.2 Changes Required to Port to New Multicore Devices

To map this code to additional CPUs, the only change required is in the constant definitions for node and port numbers in the creation of endpoints.

5.4 Multimedia Processing Use Cases

Multimedia processing includes coding and decoding between various audio and video formats. Applications range from low-power mobile devices such as cell phones, with limited resolution and audio quality, to set-top-boxes and HDTV with extremely demanding performance requirements. The following will review some of the use cases and communications characteristics for the multimedia application domain.

5.4.1 Characteristics

5.4.1.1 Simple Scenario

Figure 6 shows a simple multicore multimedia architecture. In this scenario, a multicore processor is executing a multimedia application, which is accelerated by a DSP integrated into the multicore device. The application has some data (for example, a video frame encoded in the MPEG4 format), and uses the DSP to decode into an image suitable for display. The data is moved (at least conceptually) to the DSP, the DSP processing runs, and the data is moved back to the general-purpose processor (GPP).

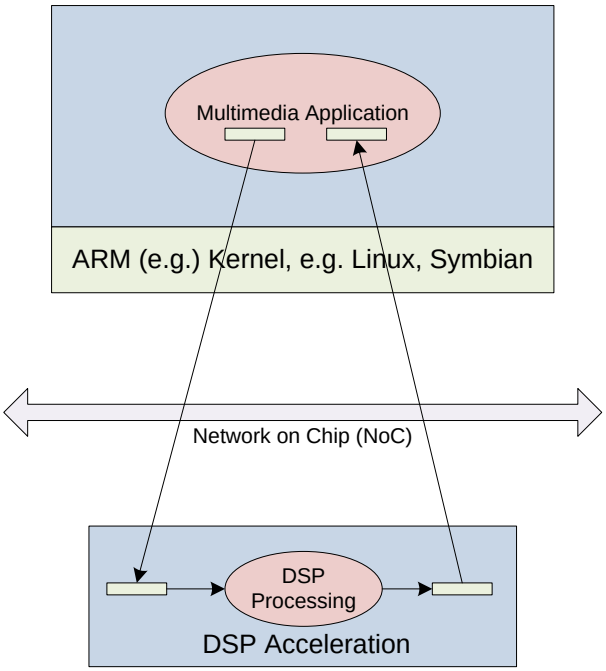


Figure 6. Simple Multimedia Scenario

Despite the simplicity of the above scenario, a number of characteristics are illustrated:

Characteristic 1: Heterogeneous Processors

The DSP and GPP have different instruction sets and potentially different data representations. The accelerators are typically 16-, 24-, or 32-bit special-purpose devices. The accelerators may have limited code and data spaces. For example, total DSP code space for the application, operating system and communication infrastructure may be under 64K instruction words. In order to save room for applications, the communication-infrastructure code footprint is ideally quite small (e.g., less than 1K VLIW instruction words). The communication data footprint is of the same magnitude.

There seems to be a trend towards 32-bit processing and less-constrained acceleration devices in the future, although this is not altogether clear, in particular for low-power mobile devices.

In the example, the application is executing in user mode on a general-purpose operating system such as Linux. Other operating systems such as QNX Neutrino, VxWorks, or WinCE may be used. The DSP is running potentially another standard operating system, or a home-grown operating system, or perhaps in a bare metal environment with no operating system.

Characteristic 2: Heterogeneous Communication Infrastructure

Figure 6 shows a network on-chip connection between the GPP (in this example an ARM processor) and DSP. This may range from a simple bus to a well-designed network-on-chip infrastructure. However, (and perhaps more typically today), this could be an ad-hoc collection of buses and bridges, with different DMA engines and other mechanisms for moving data around the system.

Characteristic 3: Heterogeneous Application Environments

Multimedia applications are very complex, and have a number of frameworks such as gstreamer, DirectShow, MDF, and OpenMax. The communication infrastructure should play well with these environments.

For example,
Figure 7 shows the OpenMax multimedia framework from the OpenMax specification[3]. Host-side communication APIs may already be defined by the framework. The MCAPI communication infrastructure is more likely suited to the tunneled communication between acceleration components.

[3] See http://www.khronos.org/openmax/

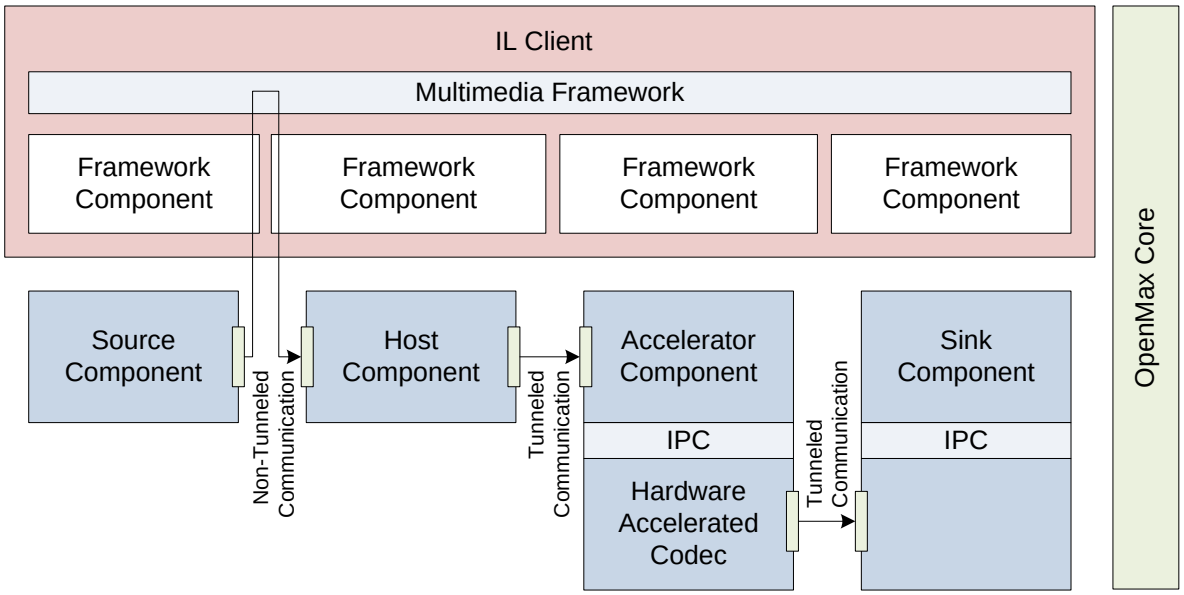


Figure 7. OpenMax Communication

Characteristic 4: Application Non-Resident Code Size

Due to the large number of standards, encoding formats, processing variants, etc, the non-resident multimedia code size is quite large (millions of lines of code). This is only the code running on the accelerators, and it does not include the code on the general-purpose processors. Of course, depending on the operating mode, only a small subset of this code will be configured and running at a particular time.

Characteristic 5: Client/Server Communication Pattern

In simple configurations as illustrated in Figure 6, the communication follows a client/server pattern. However, as we will see shortly, this is not typically the case.

In the simple case, communication may be considered coarse-grained, in that a large chunk of data is passed to the accelerator (for example, a video frame), and the DSP runs for a relatively long time to compute the results. Again, this coarse-grained behavior is not always the case, as is described below.

Characteristic 6: Coarse Grain Data Rates and Latencies

For the coarse-grained scenario, data rates and latencies are relatively conservative. Order of magnitude data sizes are 1K bytes, and rates are under 100 Hz.

Characteristic 7: Potential for Zero Copy

Performance and power are key constraints. Unnecessary data movement and/or processing should be avoided where possible. Therefore, although Figure 6 shows distinct data buffers on the GPP and DSP side, in practice the data may reside in a common shared memory. No data is actually copied when processing control is transferred from the GPP to the DSP and back again.

However, in many cases, the architecture does not have a shared-memory capability, in which case the data must be moved. Ideally, the communication infrastructure should mask these platform-dependant constraints, and allow zero copy where supportable by the architecture. This permits portable application code.

5.4.1.2 More Complex Scenarios

Figure 8 is a conceptual representation of a more complex scenario. This is motivated by applications such as H.264 encoding and decoding.

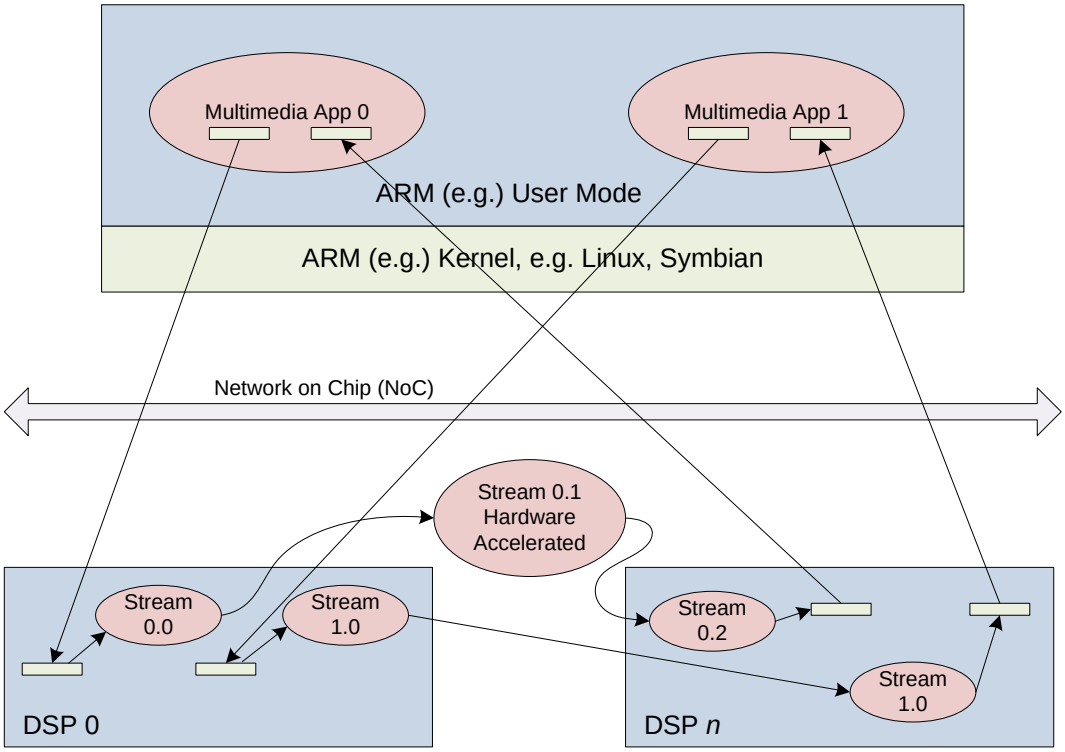


Figure 8. More Complex Scenarios

Characteristic 8: High Processing Rates

The system computation requirements may exceed 100 GOPS/sec. The following characteristic may be viewed as a corollary:

Characteristic 9: Multi-Stage Processing

Due to extreme compute requirements, no one core can perform all of the processing. Instead, processing must be split into several stages. Currently, multimedia devices with order-10 cores are common.

Characteristic 10: Hardware Components

Even when split into multiple stages, computation for a particular stage may exceed the capabilities of a programmable device. In this case, the processing stage may be implemented in hardware. Therefore, the communication infrastructure must support communication to and from hardware devices.

Characteristic 11: Trend to Finer-Grained Processing and Communication

Several factors are influencing a trend to finer-grained processing and communication. One factor is the complexity, change, and customization options of the emerging standards. This requires great flexibility in the implementation, which in turn favors an approach which decomposes the problem into many parts, with the option of quickly changing the implementation of the parts.

A natural fallout of this finer-grained decomposition is the requirement for finer-grained communication sizes and rates. For example: a processor sends 100 bytes to a hardware block for processing that takes 100 cycles. However, many different variants and approaches are in active investigation, the optimal strategy is not clear.

Characteristic 12: Streamed Processing

In a multi-stage configuration, a dataflow or streaming style of computation may be more appropriate. Depending on the granularity of the decomposition, the tokens exchanged between stages may be large (e.g., video frames), down to individual data values.

Characteristic 13: Multiple Flows

In many cases, multiple flows must be supported. For example, one or more audio streams, the main video stream, picture-in-picture streams, etc.

Characteristic 14: Dynamic Operation

Multimedia devices are increasingly dynamic. This includes many operating modes (voice calls, digital still camera, video reception and transmission, audio playback and record, etc). Arbitrary subsets of these operating modes may be selected for simultaneous operation by the user at fairly fine time-scales (e.g., in the order of seconds). Within a mode, operation may be dynamic at even finer time scales. For example, packets from a wireless connection may arrive in bursts.

Characteristic 15: Non-Deterministic Communication Rates

In many cases, the amount of data consumed and produced by a processing stage may not be deterministic, resulting in variable communication rates.

Characteristic 16: Low Power Operation

Power is a central concern for mobile devices. Components must be switched off or moved to lower voltage/operating frequency depending on system load. Computation occasionally must be redistributed over the available resources in order to optimize power consumption.

Characteristic 17: Emergent System-Level Properties and Problems

Due to the complexity of the device and the unintended coupling of components, multimedia devices may exhibit difficult-to-understand emergent properties. For example, the system may deadlock, overflow buffers, miss deadlines, or provide bursty output, etc. Mechanisms to deal with these problems are very desirable.

Characteristic 18: Quality of Service

Device operation must meet quality-of-service constraints (processing latency, throughput, jitter, etc). Ideally, the communications infrastructure should support this. Due to high-performance requirements, parts of this infrastructure may be in hardware.

Characteristic 19: Security

Multimedia flows have a strong security requirement. It may be required to prevent access to decoded data (for example), even if the operating system(s) have been compromised.

5.4.1.3 **Metrics**

Table 1 summarizes some of the quantitative characteristics of the multimedia application domain.

Table 1. Characteristics of the Multimedia Application Domain

Attribute	Value (ranges)	Notes
Communication code size footprint on accelerators	8 K bytes	Becoming less constrained in the future
Communication data size footprint	8 K bytes	Becoming less constrained in the future
Non-resident code size	millions of lines of code	Increasing in the future
Number of processors	order 10	
Hardware accelerators	under 10	Different accelerators in high-end systems
Communication sizes	1K to under 100 bytes	Trend to finer-grain
Communication latency	10 mSec to under 1 uSec	Trend to finer-grain in future

5.4.2 Key Functionality Requirements

Table 2 summarizes the key requirements of multimedia devices and comments upon the ability MCAPI to meet the requirement.

Table 2. Multimedia Derived Requirements

Description	Derived From	MCAPI
Low footprint on constrained devices (code and data size)	Characteristic 1	Yes
Framework for heterogeneous data types	Characteristic 1	Yes
Framework for complex transport configuration, quality of service, message delivery semantics and queuing policies (FIFO, most recent only, etc)	Characteristic 2	Some, others can be built on top.
Minimal assumptions about operating environment, and services required	Characteristic 3	Yes
No unnecessary data movement if architecture has supporting features (e.g., shared memory)	Characteristic 7	Yes, zero copy functionality
Allow direct binding to hardware, allowing end-to-end message delivery rates in the order of 100 cycles	Characteristic 8 Characteristic 9 Characteristic 10 Characteristic 11	Yes, no MCAPI feature prevents it
Framework for word-by-word message send/receive	Characteristic 8 Characteristic 9 Characteristic 10 Characteristic 11 Characteristic 12	Yes
Allow blocking and non-blocking operations	Characteristic 3	Yes
Provide message matching framework, to allow selection of messages of particular type, priority, deadline, etc	Characteristic 18	No, can be built on top of MCAPI.
Allow multi-drop messages. E.g., potential for >1 senders to queuing point, >1 readers	Characteristic 16 Characteristic 18	No, can be built on top of MCAPI.

5.4.3 Pseudocode Example

The pseudocode scenario contains the execution of a DSP algorithm, such as a FIR filter, on a multicore processor system containing a general-purpose processor (GPP) and DSP. The high-level steps are:

1. Initialize communication channels between processors.
2. GPP sends data for processing to DSP.
3. GPP sends code to execute to DSP
4. GPP signals for the DSP to begin execution.
5. GPP retrieves processed data from DSP

```
/*
 * MCAPI 2.000 Multimedia Use Case
 * shared.h
 *
 */

#ifndef SHARED_H_
#define SHARED_H_

#define CHECK_STATUS(status)
/* Shared Header File, shared.h */
enum {
      DSP_READY,
      DSP_DATA,
      DSP_CODE,
      DSP_TERMINATE,
      DSP_EXECUTE,
      COMPLETED
};

#define DOMAIN_0        0

#define PORT_COMMAND    0
#define PORT_DATA       1
#define PORT_DATA_RECV  2

#endif /* SHARED_H_ */


/*
 * MCAPI 2.000 Multimedia Use Case
 * gp_proc.c
 *
 */

/* General Purpose Processor Code */
#include "shared.h"
#include "mcapi.h"

/* Predefined Node numbering */
#define GPP_1 0  /* 0-63 reserved for homogenous MC */
#define DSP_1 64 /* 64+ used for DSP nodes */

/* Predefined Port numbering */
#define PORT_COMMAND 0
#define PORT_DATA 1
```

```
#define PORT_DATA_REC 2

mcapi_endpoint_t command_endpoint;
mcapi_endpoint_t data_endpoint;
mcapi_endpoint_t remote_command_endpoint;
mcapi_endpoint_t remote_data_endpoint;
mcapi_endpoint_t data_recv_endpoint;
mcapi_endpoint_t remote_data_recv_endpoint ;

mcapi_pktchan_send_hndl_t data_chan;
mcapi_sclchan_send_hndl_t command_chan;
mcapi_pktchan_recv_hndl_t data_recv_chan;

void *data;
int data_size;
void *code;
int code_size;

void initialize_comms()
{
      mcapi_node_t gp_node = 0;
      mcapi_node_attributes_t mcapi_node_attributes;
      mcapi_param_t mcapi_parameters;
      mcapi_info_t mcapi_info;
      mcapi_status_t status;
      mcapi_request_t request;
      mcapi_timeout_t timeout = 500;


      mcapi_node_init_attributes(&mcapi_node_attributes, &status);

      mcapi_initialize(DOMAIN_0, gp_node, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &status);
      CHECK_STATUS(status);

      command_endpoint = mcapi_endpoint_create(PORT_COMMAND, & status);
      CHECK_STATUS(status);

      data_endpoint = mcapi_endpoint_create(PORT_DATA, & status);
      CHECK_STATUS(status);

      data_recv_endpoint = mcapi_endpoint_create(PORT_DATA_RECV, & status);
      CHECK_STATUS(status);

      remote_command_endpoint = mcapi_endpoint_get(DOMAIN_0, DSP_1,
             PORT_COMMAND, timeout, &status);
      CHECK_STATUS(status);

      remote_data_endpoint = mcapi_endpoint_get(DOMAIN_0, DSP_1, PORT_DATA,
             timeout, &status);
      CHECK_STATUS(status);

      remote_data_recv_endpoint = mcapi_endpoint_get(DOMAIN_0, DSP_1,
             PORT_DATA_RECV, timeout, &status);
      CHECK_STATUS(status);

      mcapi_pktchan_connect_i(data_endpoint, remote_data_endpoint, &request,
            &status);
      CHECK_STATUS(status);
```

```
      mcapi_sclchan_connect_i(command_endpoint, remote_command_endpoint,
            &request, &status)
      CHECK_STATUS(status);

      mcapi_pktchan_connect_i(data_recv_endpoint, remote_data_recv_endpoint,
            &request, &status);
      CHECK_STATUS(status);

      mcapi_pktchan_send_open_i(&data_chan, data_endpoint, &request,
             &status);
      CHECK_STATUS(status);

      mcapi_sclchan_send_open_i(&command_chan, command_endpoint, &request,
            &status);
      CHECK_STATUS(status);

      mcapi_pktchan_recv_open_i(&data_recv_chan, data_recv_endpoint,
             &request, &status);
  CHECK_STATUS(status);

}

void send_data(void *data, int size)
{
      mcapi_status_t status;
      mcapi_pktchan_send(data_chan, data, size, &status);
      CHECK_STATUS(status);
}

void send_dsp_cmd(int cmd)
{
      mcapi_status_t status;
      mcapi_sclchan_send_uint32(command_chan, cmd, &status);
      CHECK_STATUS(status);
}

void read_data(void **dst, int *size)
{
      mcapi_status_t status;
      mcapi_pktchan_recv(data_recv_chan, dst, (size_t)size, &status);
      CHECK_STATUS(status);
}

void shutdown_comms()
{
      mcapi_status_t status;
      mcapi_request_t request;

      mcapi_pktchan_recv_close_i(data_recv_chan, &request, &status);
      CHECK_STATUS(status);
      mcapi_pktchan_send_close_i(data_chan, &request, &status);
      CHECK_STATUS(status);
      mcapi_sclchan_send_close_i(command_chan, &request, &status);
      CHECK_STATUS(status);

      mcapi_endpoint_delete(command_endpoint, &status)
            CHECK_STATUS(status);

      mcapi_endpoint_delete(data_endpoint, &status)
            CHECK_STATUS(status);
```

```
	mcapi_endpoint_delete(data_recv_endpoint, &status)
	CHECK_STATUS(status);
}

void *allocate(int size) {
	/* Routine to allocate memory */
}

int perform_multimedia_function(void *code, int code_size, void *data,
		int data_size)
{
	void* result;
	int size = 0;
	initialize_comms();
	send_dsp_cmd(DSP_DATA);
	send_data(data, data_size);
	send_dsp_cmd(DSP_CODE);
	send_data(code, code_size);
	send_dsp_cmd(DSP_EXECUTE);
	read_data(&result, &size);
	send_dsp_cmd(DSP_TERMINATE);
	shutdown_comms();
	return COMPLETED;
}


/*
 * MCAPI 2.000 Multimedia Use Case
 * dsp.c
 *
 */

/* DSP Code */

#include "shared.h"
#include "mcapi.h"


mcapi_endpoint_t dsp_command_endpoint;
mcapi_endpoint_t dsp_data_endpoint;
mcapi_endpoint_t dsp_data_send_endpoint;

mcapi_pktchan_recv_hndl_t data_chan;
mcapi_sclchan_recv_hndl_t command_chan;
mcapi_pktchan_send_hndl_t data_send_chan;

void *buffer;
void *code_buffer;
void *data_buffer;
size_t code_size;
size_t data_size;
void *result_buffer;
int result_size;

int dsp_initialize()
{
	mcapi_node_t dsp_node = 0;
	mcapi_node_attributes_t mcapi_node_attributes;
	mcapi_param_t mcapi_parameters;
	mcapi_info_t mcapi_info;
```

```
      mcapi_status_t status;

      mcapi_request_t request;

  mcapi_node_init_attributes(&mcapi_node_attributes, &status);

      mcapi_initialize(DOMAIN_0, dsp_node, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &status);

      CHECK_STATUS(status);

      dsp_command_endpoint = mcapi_endpoint_create(PORT_COMMAND, &status);
      CHECK_STATUS(status);

      dsp_data_endpoint = mcapi_endpoint_create(PORT_DATA, &status);
      CHECK_STATUS(status);
      dsp_data_send_endpoint = mcapi_endpoint_create(PORT_DATA_RECV,
            &status);
      CHECK_STATUS(status);

      mcapi_pktchan_recv_open_i(&data_chan, dsp_data_endpoint, &request,
           &status);
      CHECK_STATUS(status);

      mcapi_sclchan_recv_open_i(&command_chan, dsp_command_endpoint,
            &request, &status);
      CHECK_STATUS(status);

      mcapi_pktchan_send_open_i(&data_send_chan, dsp_data_send_endpoint,
            &request, &status);
      CHECK_STATUS(status);

      return status;
}

int dsp_shutdown()
{
      mcapi_status_t status;
      mcapi_request_t request;

      mcapi_pktchan_release(data_buffer, &status);

      mcapi_pktchan_release(code_buffer, &status);

      mcapi_pktchan_recv_close_i(data_chan, &request, &status);
      CHECK_STATUS(status);

      mcapi_sclchan_recv_close_i(command_chan, &request, &status);
      CHECK_STATUS(status);

      mcapi_pktchan_send_close_i(data_send_chan, &request, &status);
      CHECK_STATUS(status);

      mcapi_endpoint_delete(dsp_command_endpoint, &status);
      CHECK_STATUS(status);

      mcapi_endpoint_delete(dsp_data_endpoint, &status);
      CHECK_STATUS(status);

      mcapi_endpoint_delete(dsp_data_send_endpoint, &status);
      CHECK_STATUS(status);
```

```
      return status;
}

int receive_dsp_cmd()
{
      mcapi_uint32_t cmd;
      mcapi_status_t status;

      cmd = mcapi_sclchan_recv_uint32(command_chan, &status);
      CHECK_STATUS(status);
}

int dsp_command_loop()
{
  int command = DSP_READY;
  mcapi_status_t status;

  dsp_initialize();

  while (command != DSP_TERMINATE) {
    switch (command) {
      case DSP_DATA:
        mcapi_pktchan_recv(data_chan, &data_buffer, &data_size, &status);
        CHECK_STATUS(status);
      break;
      case DSP_CODE:
        mcapi_pktchan_recv(data_chan, &code_buffer, &code_size, &status);
        CHECK_STATUS(status);
        /* Copy code to from buffer to local execute memory */
      break;
      case DSP_EXECUTE:
        /* Tell DSP to Execute - Assume code writes to result_buffer */

        mcapi_pktchan_send(data_send_chan, result_buffer, result_size,
            &status);
        CHECK_STATUS(status);
      break;
      case DSP_TERMINATE:
        dsp_shutdown();
      break;
      default:
      break;
    }
    command = receive_dsp_cmd();
  }
}
```

5.5 Packet Processing

This example presents the typical startup and inner loop of a packet processing application. There are two source files: `load_balancer.c` and `worker.c`. The main entry point is in `load_balancer.c`.

The program begins in the load balancer, which spawns a set of worker processes and binds channels to them. Each worker has two channel connections to the load balancer: a packet channel for work requests and a scalar channel for acks. When work arrives on the packet channel from the load balancer, the worker processes it and then sends back an ack to the load balancer. The load balancer will not send new work to a worker unless an ack word is available.

This example uses both packet channels and scalar channels. Scalar channels are used for acks from the workers to the load balancer, because each ack is a single word and performance will be better without the packetization overhead. Packet channels are used to send work requests to the workers, because each work request is a structure containing multiple fields. The example also shows how to use both statically named and dynamic (anonymous) endpoints. It uses the endpoint creation functions `mcapi_endpoint_create()`, `mcapi_create_anonymous_endpoint()`, and `mcapi_endpoint_get()`.

On architectures with hardware support for scalars, it would make more sense to use scalar channels for the work requests also. Work requests are usually composed of only a few words, so they can be sent as individual words. More importantly, the load balancer is the rate-limiting step in this program; we can always add more workers but they are useless if the load balancer cannot feed them. Since the load balancer is limited by communication overhead, hardware-accelerated scalar channels could potentially improve overall performance if the overhead of a packet-channel send is greater than the comparable scalar-channel sends.

5.5.1 Packet Processing Code

5.5.1.1 common.h

```
/*
 * MCAPI 2.000 Packet Processing Use Case
 * common.h
 *
 */

// Header file containing a couple of declarations that are shared
// between the load balancer and the worker.

#ifndef COMMON_H_
#define COMMON_H_

#include <stdio.h>
#include <stdlib.h>

extern char* mcapi_strerror(int s);

enum {
  WORKER_REQUEST_PORT_ID,
  WORKER_ACK_PORT_ID
};

#define DOMAIN_0  0
#define NODE_LOAD_BALANCER 0

#define CHECK_STATUS(S) do {  \
  if (S != MCAPI_SUCCESS) {    \
    printf("Failed with MCAPI status %d (%s)\n", S, mcapi_strerror(S)); \
    exit(1); \
 }} while (0)

typedef struct{
      unsigned char is_valid;
      int worker;
}     PacketInfo;

#endif /* COMMON_H_ */
```

```
/*
 * MCAPI 2.000 Packet Processing Use Case
 * load_balancer.c
 *
 */

#include "common.h"
#include "mcapi.h"

#include <stdbool.h>

extern void spawn_new_thread(void* funct, int worker_id);
extern void worker_spawn_function(int worker_num);
extern int get_next_packet(PacketInfo* packet_info);
extern void drop_packet(PacketInfo* packet_info);

// We will have four worker processes
#define NUM_WORKERS 4

// An array of packet channel send ports for sending work descriptors
// to each of the worker processes.
mcapi_port_t work_requests_out_port[NUM_WORKERS];

// An array of scalar channel receive ports for getting acks back from
// the workers.
mcapi_port_t acks_in_port[NUM_WORKERS];

mcapi_pktchan_send_hndl_t work_requests_out_hndl[NUM_WORKERS];
mcapi_pktchan_recv_hndl_t acks_in_hndl[NUM_WORKERS];


// function declarations
void create_and_init_workers(void);
void dispatch_packets(void);
void shutdown_lb(void);

// The entrypoint for this packet processing application.
int main(void)
{
  mcapi_node_attributes_t mcapi_node_attributes;
      mcapi_param_t mcapi_parameters;
      mcapi_info_t mcapi_info;
      mcapi_status_t status;


  mcapi_node_init_attributes(&mcapi_node_attributes, &status);

      mcapi_initialize(DOMAIN_0, NODE_LOAD_BALANCER, &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &status);

  create_and_init_workers();
  dispatch_packets();
  shutdown_lb();

  return 0;
}


void create_and_init_workers()
{
      int i;
```

```
      mcapi_request_t request;

  for (i = 0; i < NUM_WORKERS; i++)
  {
    mcapi_timeout_t timeout = 500;
    mcapi_status_t status;

    // Spawn a new thread; pass parameters so the new thread will execute
    // worker_spawn_function(bootstrap_endpoint)
    spawn_new_thread(&worker_spawn_function, i);
    int worker_node = i + 1;

    // Create a send endpoint to send packets to the worker; get the
    // worker's receive endpoint via mcapi_endpoint_get()
    mcapi_endpoint_t work_request_out_endpoint =
            mcapi_endpoint_create(MCAPI_PORT_ANY, &status);
    CHECK_STATUS(status);
    mcapi_endpoint_t work_request_remote_endpoint =
            mcapi_endpoint_get(DOMAIN_0, worker_node,
             WORKER_REQUEST_PORT_ID, timeout, &status);
    CHECK_STATUS(status);

    // Bind the channel and open our local send port.
            mcapi_pktchan_connect_i(work_request_out_endpoint,
            work_request_remote_endpoint, &request, &status);
    CHECK_STATUS(status);
    mcapi_pktchan_send_open_i(&work_requests_out_hndl[i],
            work_request_out_endpoint, &request, &status);
    CHECK_STATUS(status);


    // Repeat the process to create an ack scalar channel from the
    // worker back to us.
    mcapi_endpoint_t ack_in_endpoint = mcapi_endpoint_create(MCAPI_PORT_ANY,
             &status);
    CHECK_STATUS(status);
    mcapi_endpoint_t ack_remote_endpoint =
            mcapi_endpoint_get(DOMAIN_0, worker_node, WORKER_ACK_PORT_ID,
            timeout, &status);
    CHECK_STATUS(status);

    mcapi_sclchan_connect_i(ack_remote_endpoint, ack_in_endpoint, &request,
            &status);
    CHECK_STATUS(status);

    mcapi_sclchan_recv_open_i(&acks_in_hndl[i], ack_in_endpoint, &request,
            &status);
    CHECK_STATUS(status);
  }
}


void dispatch_packets()
{
  PacketInfo packet_info;
  mcapi_status_t status;

  while(get_next_packet(&packet_info))
  {
    // Because we maintain "session state" across packets, each
    // incoming packet is associated with a particular worker.
```

```
    int worker = packet_info.worker;

    // Each worker sends back acks when it is ready for more work.
    if (mcapi_sclchan_available(acks_in_hndl[worker], &status))
    {
      // An ack is available; pull it off and send more work
      int unused_result = mcapi_sclchan_recv_uint8(acks_in_hndl[worker],
            &status);
      mcapi_pktchan_send(work_requests_out_hndl[worker], &packet_info,
            sizeof(packet_info), &status);
      CHECK_STATUS(status);
    }
    else
    {
      // No ack available; drop the packet (or queue, or do some other
      // form of exception processing) and move on.
      drop_packet(&packet_info);
    }
  }
}


void shutdown_lb()
{
      mcapi_status_t status;
      mcapi_request_t request;
  int i;

  // Shutdown each worker in turn
  for(i = 0; i < NUM_WORKERS; i++)
  {
    // Send an "invalid" packet to trigger worker shutdown
    PacketInfo invalid_work;
    invalid_work.is_valid = false;
    mcapi_pktchan_send(work_requests_out_hndl[i], &invalid_work,
            sizeof(invalid_work), &status);
    CHECK_STATUS(status);

    // Close our ports; don't worry about errors (what would we do?)
    mcapi_pktchan_send_close_i(work_requests_out_hndl[i], &request,
            &status);
    mcapi_sclchan_recv_close_i(acks_in_hndl[i], &request, &status);
  }
}



/*
 * MCAPI 2.000 Packet Processing Use Case
 * worker.c
 *
 */

#include "common.h"
#include "mcapi.h"
#include <stdbool.h>

void some_function_or_another(PacketInfo* packet_info);

// Local port for incoming work requests from the load balancer.  We
// use a packet channel because each work request is a structure with
```

```
// multiple fields specifying the work to be done.
mcapi_port_t work_request_in;

// Local port for outgoing acks to the load balancer.  We use a scalar
// channel because each ack is a single word indicating the number of
// work items that have been completed.
mcapi_port_t ack_out;

// a local buffer for use by the incoming packet channel
char work_channel_buffer[1024];


// function declarations
void bind_channels(void);
void do_work(void);
void shutdown(void);

// The function called within each new worker thread.  This function
// binds takes an endpoint in the load balancer as its parameter so
// that it can communicate with the load balancer and create channels
// between the worker and the load balancer.
void worker_spawn_function(int worker_num)
{
  mcapi_node_attributes_t mcapi_node_attributes;
      mcapi_param_t mcapi_parameters;
      mcapi_info_t mcapi_info;
      mcapi_status_t status;

  mcapi_node_init_attributes(&mcapi_node_attributes, &status);

      mcapi_initialize(DOMAIN_0, (1 + worker_num), &mcapi_node_attributes,
            &mcapi_parameters, &mcapi_info, &status);

  bind_channels();
  do_work();
  shutdown();
}

void bind_channels()
{
      mcapi_status_t status;
      mcapi_request_t request;
      mcapi_pktchan_recv_hndl_t work_request_in;
      mcapi_sclchan_send_hndl_t ack_out;

  // Create a packet receive endpoint for incoming work requests; use
  // a static port number so that load balancer can look it up via
  // mcapi_endpoint_get().
  mcapi_endpoint_t work_request_in_endpoint =
            mcapi_endpoint_create(WORKER_REQUEST_PORT_ID, &status);
  CHECK_STATUS(status);

  // The load balancer will bind a channel to that endpoint; we use
  // the open function to synchronize and initialize our local packet
  // receive port.
  mcapi_pktchan_recv_open_i(&work_request_in, work_request_in_endpoint,
            &request, &status);
  CHECK_STATUS(status);

  // Repeat the process to create a scalar channel from us back to the
  // load balancer.
```

```
  mcapi_endpoint_t ack_out_endpoint =
            mcapi_endpoint_create(WORKER_ACK_PORT_ID, &status);
  CHECK_STATUS(status);

  mcapi_sclchan_send_open_i(&ack_out, ack_out_endpoint, &request, &status);
  CHECK_STATUS(status);
}


void do_work()
{
  mcapi_status_t status;
  size_t size;

  // The first thing we do is send an ack so that the load balancer
  // knows we're ready for work.
  mcapi_sclchan_send_uint8(ack_out, 1 /* Any value would do. */, &status);

  while (1)
  {
    // receive a work request from the load balancer
    PacketInfo* work_info;
    mcapi_pktchan_recv(work_request_in, (void*)&work_info, &size, &status);
    CHECK_STATUS(status);

    // if the work request is marked "invalid", we're done
    if (work_info->is_valid == false)
    {
      break;
    }

    // do the work
    some_function_or_another(work_info);

    // we're done; return the buffer and send an ack
    mcapi_pktchan_release(work_info, &status);
    mcapi_sclchan_send_uint8(ack_out, 1 /* Any value would do. */, &status);
  }
}


void shutdown()
{
  mcapi_status_t status;
      mcapi_request_t request;

  // close our channels
  mcapi_pktchan_recv_close_i(work_request_in, &request, &status);
  mcapi_sclchan_send_close_i(ack_out, &request, &status);
}
```

6. Appendix A: Acknowledgements

The MCAPI working group would like to acknowledge the significant contributions of the following people in the creation of this API specification:

Working Group

Ajay Kamalvanshi, Nokia Siemens Networks
Anant Agarwal, Tilera
Sven Brehmer, PolyCore Software (chair)
Max Domeika, Intel
Peter Flake, Imperas
Steven Furr, Freescale Semiconductor, Inc.
Masaki Gondo, eSOL
Patrick Griffin, Tilera
Jim Holt, Freescale Semiconductor, Inc.
Rob Jackson, Imperas
Kevin Kissell, MIPS
Maarten Koenig, Wind River
Markus Levy, Multicore Association
Charles Pilkington, ST Micro
Andrew Richards, Codeplay
Colin Riley, Codeplay
Frank Schirrmeister, Imperas

The MCAPI working group also would like to thank the external reviewers who provided input and helped us to improve the specification below is a partial list of the external reviewers (some preferred to not be mentioned).

Reviewers

David Addison, ST Micro
Sterling Augustine, Tensilica
Badrinath Dorairajan, Aricent
Marc Gauthier, Tensilica
David Heine, Tensilica
Dominic Herity, Redplain Technology
Arun Joseph, IBM
Tammy Leino, Mentor Graphics
Michael Kardonik, Freescale Semiconductor, Inc.
Anjna Khanna, Cadence Design Systems
Grant Martin, Tensilica
Tim Mattson, Intel
Dror Maydan, Tensilica
Girish Mundada, Micronas
Karl Mörner, Enea
Emeka Nwafor, Zeligsoft
Don Padgett, Padgett Computing
Ola Redell, Enea
Michele Reese, Freescale Semiconductor, Inc.
Greg Regnier, Intel

Vedvyas Shanbhogue, Intel
Patrik Strömblad, Enea

7. Appendix B: Header Files

7.1 mca.h

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mca.h
 *
 * Version 2.015, March 2011
 *
 */

#ifndef MCA_H
#define MCA_H

/*
 * The mca_impl_spec.h header file is vendor/implementation specific,
 * and should contain declarations and definitions specific to a particular
 * implementation.
 *
 * This file must be provided by each implementation.  It is recommended
 * that these types be either pointers or 32 bit scalars, allowing simple
 * arithmetic equality comparison (a == b).
 * Implementers may which of these type are used.
 *
 * It MUST contain type definitions for the following types.
 *
 * mca_request_t;
```

```
 *
 */
#include "mca_impl_spec.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * MCA type definitions
 */
typedef int                   mca_int_t;
typedef char                  mca_int8_t;
typedef short                 mca_int16_t;
typedef int                   mca_int32_t;
typedef long long             mca_int64_t;
typedef unsigned int          mca_uint_t;
typedef unsigned char         mca_uint8_t;
typedef unsigned short        mca_uint16_t;
typedef unsigned int          mca_uint32_t;
typedef unsigned long long    mca_uint64_t;
typedef unsigned char         mca_boolean_t;
typedef unsigned int          mca_node_t;
typedef unsigned int          mca_status_t;
typedef unsigned int          mca_timeout_t;
typedef unsigned int          mca_domain_t;

/* Constants */
#define MCA_TRUE                1
#define MCA_FALSE               0
#define MCA_NULL                0         /* MCA Zero value */
#define MCA_INFINITE           (~0)  /* Wait forever, no timeout */

/* In/out parameter indication macros */
#ifndef MCA_IN
#define MCA_IN const
#endif /* MCA_IN */

#ifndef MCA_OUT
#define MCA_OUT
#endif /* MCA_OUT */

/* Alignment macros */
#ifdef __GNUC__
#define MCA_DECL_ALIGNED __attribute__ ((aligned (32)))
#else
#define MCA_DECL_ALIGNED /* MCA_DECL_ALIGNED alignment macro currently only
                            supports GNU compiler */
#endif /* __GNUC__ */

#ifndef MCA_BUF_ALIGN
#define MCA_BUF_ALIGN
#endif

/*
 * MCA organization id's (for assignment of organization specific attribute
 * numbers)
 */
#define MCA_ORG_ID_PSI 0      /* PolyCore Software, Inc. */
#define MCA_ORG_ID_FSL 1      /* Freescale, Inc. */
#define MCA_ORG_ID_MGC 2      /* Mentor Graphics, Corp. */
```

```
#define MCA_ORG_ID_TBA 3      /* To be assigned */
/* And so forth */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCA_H */
```

7.2 mcapi.h

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mcapi.h
 *
 * Version 2.015, March 2011
*/

#ifndef MCAPI_H
#define MCAPI_H

#include <stddef.h>                              /* Required for size_t */
#include "mca.h"
/*
 * The mcapi_impl_spec.h header file is vendor/implementation specific,
 * and should contain declarations and definitions specific to a particular
 * implementation.
 *
 * This file must be provided by each implementation.
 *
 * It MUST contain type definitions for the following types, which must be
```

```
 * either pointers or 32 bit scalars, allowing simple arithmetic equality
 * comparison (a == b).
 * Implementers may which of these type are used.
 *
 * mcapi_endpoint_t;                Note: The endpoint identifier must be
 *                                                topology unique.
 * mcapi_pktchan_recv_hndl_t;
 * mcapi_pktchan_send_hndl_t;
 * mcapi_sclchan_send_hndl_t;
 * mcapi_sclchan_recv_hndl_t;
 *
 *
 * It MUST contain the following definition:
 * mcapi_param_t;
 *
 * It MUST contain the following definitions:
 *
 * Number of MCAPI reserved ports, starting at port 0. Reserved ports can
 * be used for implementation specific purposes.
 *
 * MCAPI_NUM_RESERVED_PORTS   1     Number of reserved ports
 *                                  starting at port 0
 *
 * Implementation defined MCAPI MIN and MAX values.
 *
 * Implementations may parameterize implementation specific max values,
 * smaller that the MCAPI max values. Implementations must specify what
 * those smaller values are and how they are set.
 *
 * MCAPI_MAX_DOMAIN           Maximum value for domain
 * MCAPI_MAX_NODE             Maximum value for node
 * MCAPI_MAX_PORT             Maximum value for port
 * MCAPI_MAX_MESSAGE_SIZE     Maximum message size
 * MCAPI_MAX_PACKET_SIZE      Maximum packet size
 *
 * Implementations may parameterize implementation specific priority min
 * value and set the number of reserved ports. Implementations must specify
 * what those values are and how they are set.
 *
 * MCAPI_MIN_PORT             Minimum value for port
 * MCAPI_MIN_PRORITY          Minimum priority value
 *
 */
#include "mcapi_impl_spec.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * MCAPI type definitions
 * (Additional typedefs under the attribute and initialization sections
 * below)
 */
typedef mca_int_t       mcapi_int_t;
typedef mca_int8_t      mcapi_int8_t;
typedef mca_int16_t     mcapi_int16_t;
typedef mca_int32_t     mcapi_int32_t;
typedef mca_int64_t     mcapi_int64_t;
typedef mca_uint_t      mcapi_uint_t;
typedef mca_uint8_t     mcapi_uint8_t;
```

```
typedef mca_uint16_t    mcapi_uint16_t;
typedef mca_uint32_t    mcapi_uint32_t;
typedef mca_uint64_t    mcapi_uint64_t;
typedef mca_boolean_t   mcapi_boolean_t;
typedef mca_domain_t    mcapi_domain_t;
typedef mca_node_t      mcapi_node_t;
typedef unsigned int    mcapi_port_t; typedef mca_status_t  mcapi_status_t;
typedef mca_request_t   mcapi_request_t;
typedef mca_timeout_t   mcapi_timeout_t;
typedef unsigned int    mcapi_priority_t;


/* The following constants are not implementation defined */
#define MCAPI_VERSION               2015  /* Version 2.015 (major #
                                             + minor # (3-digit)) */
#define MCAPI_TRUE      MCA_TRUE
#define MCAPI_FALSE     MCA_FALSE
#define MCAPI_NULL      MCA_NULL          /* MCAPI Zero value */
#define MCAPI_PORT_ANY  (~0)              /* Create endpoint using
                                             the next available port */
#define MCAPI_TIMEOUT_INFINITE      (~0)  /* Wait forever, no timeout */
#define MCAPI_TIMEOUT_IMMEDIATE       0   /* Return immediately, with
                                             success or failure */
#define MCAPI_NODE_INVALID          (~0)  /* Return value for invalid
                                             node */
#define MCAPI_DOMAIN_INVALID        (~0)  /* Return value for invalid
                                             domain */
#define MCAPI_RETURN_VALUE_INVALID  (~0)  /* Invalid return value */
#define MCAPI_MAX_PRORITY             0   /* Maximum priority value */
#define MCAPI_MAX_STATUS_MSG_LEN     32   /* Maximum status code message
                                             length */


/*
 * MCAPI Status codes
 */
enum mcapi_status_codes {
      MCAPI_SUCCESS = 1,            /* Indicates operation was successful */
      MCAPI_PENDING,                /* Indicates operation is pending
                                       without errors */
      MCAPI_TIMEOUT,                /* The operation timed out */
      MCAPI_ERR_PARAMETER,          /* Incorrect parameter */
      MCAPI_ERR_DOMAIN_INVALID,     /* The parameter is not a valid domain
                                       */
      MCAPI_ERR_NODE_INVALID,       /* The parameter is not a valid node */
      MCAPI_ERR_NODE_INITFAILED,    /* The MCAPI node could not be
                                       initialized */
      MCAPI_ERR_NODE_INITIALIZED,   /* MCAPI node is already initialized */
      MCAPI_ERR_NODE_NOTINIT,       /* The MCAPI node is not initialized */
      MCAPI_ERR_NODE_FINALFAILED,   /* The MCAPI could not be finalized */
      MCAPI_ERR_PORT_INVALID,       /* The parameter is not a valid port */
      MCAPI_ERR_ENDP_INVALID,       /* The parameter is not a valid
                                       endpoint descriptor */
      MCAPI_ERR_ENDP_EXISTS,        /* The endpoint is already created */
      MCAPI_ERR_ENDP_GET_LIMIT,     /* Endpoint get reference count is to
                                       high */
      MCAPI_ERR_ENDP_DELETED,       /* The endpoint has been deleted */
      MCAPI_ERR_ENDP_NOTOWNER,      /* An endpoint can only be deleted by
                                       its creator */
      MCAPI_ERR_ENDP_REMOTE,        /* Certain operations are only allowed
                                       on the node local endpoints */
      MCAPI_ERR_ATTR_INCOMPATIBLE,  /* Connection of endpoints with
```

```
                                        incompatible attributes not allowed
                                        */
      MCAPI_ERR_ATTR_SIZE,              /* Incorrect attribute size */
      MCAPI_ERR_ATTR_NUM,               /* Incorrect attribute number */
      MCAPI_ERR_ATTR_VALUE,             /* Incorrect attribute value */
      MCAPI_ERR_ATTR_NOTSUPPORTED,      /* Attribute not supported by the
                                        implementation */
      MCAPI_ERR_ATTR_READONLY,          /* Attribute is read only */
      MCAPI_ERR_MSG_SIZE,               /* The message size exceeds the maximum
                                        size allowed by the MCAPI
                                        implementation */
      MCAPI_ERR_MSG_TRUNCATED,          /* The message size exceeds the buffer
                                        size */
      MCAPI_ERR_CHAN_OPEN,              /* A channel is open, certain operations
                                        are not allowed */
      MCAPI_ERR_CHAN_TYPE,              /* Attempt to open a packet/scalar
                                        channel on an endpoint that has been
                                        connected with a different channel
                                        type */
      MCAPI_ERR_CHAN_DIRECTION,         /* Attempt to open a send handle on a
                                        port that was connected as a
                                        receiver, or vice versa */
      MCAPI_ERR_CHAN_CONNECTED,         /* A channel connection has already been
                                        established for one or both of the
                                        specified endpoints */
      MCAPI_ERR_CHAN_OPENPENDING,       /* An open request is pending */
      MCAPI_ERR_CHAN_CLOSEPENDING,      /* A close request is pending */
      MCAPI_ERR_CHAN_NOTOPEN,           /* The channel is not open (cannot be
                                        closed) */
      MCAPI_ERR_CHAN_INVALID,           /* Argument is not a channel handle */
      MCAPI_ERR_PKT_SIZE,               /* The packet size exceeds the maximum
                                        size allowed by the MCAPI
                                        implementation */
      MCAPI_ERR_TRANSMISSION,           /* Transmission failure */
      MCAPI_ERR_PRIORITY,               /* Incorrect priority level */
      MCAPI_ERR_BUF_INVALID,            /* Not a valid buffer descriptor */
      MCAPI_ERR_MEM_LIMIT,              /* Out of memory */
      MCAPI_ERR_REQUEST_INVALID,        /* Argument is not a valid request
                                        handle */
      MCAPI_ERR_REQUEST_LIMIT,          /* Out of request handles */
      MCAPI_ERR_REQUEST_CANCELLED,      /* The request was already canceled */
      MCAPI_ERR_WAIT_PENDING,           /* A wait is pending */
      MCAPI_ERR_GENERAL,                /* To be used by implementations for
                                        error conditions not covered by the
                                        other status codes */
      MCAPI_STATUSCODE_END              /* This should always be last */
};

/*
 * Node attribute numbers
 *
 */
enum mcapi_node_attribute_numbers {
      MCAPI_NODE_ATTR_TYPE,             /* Node type */
      MCAPI_NODE_ATTR_END               /* This should always be last */
};

/* MCAPI_NODE_ATTR_TYPE */
typedef mcapi_uint_t mcapi_node_attr_type_t;

/* Node Types */
```

```
enum mcapi_node_attribute_types {
      MCAPI_NODE_ATTR_TYPE_REGULAR, /* Default - Regular node type */
      MCAPI_NODE_ATTR_TYPE_END      /* This should always be last */
};

/*
 * Endpoint attribute numbers
 *
 * Some of the endpoint attributes must have the same value for two
 * endpoints to be connected by a channel. Those attributes are identified
 * below with "Channel compatibility attribute"
 *
 */
enum mcapi_endp_attribute_numbers {
      MCAPI_ENDP_ATTR_MAX_PAYLOAD_SIZE,    /* Maximum payload size -
                                              Channel compatibility attribute
                                              */
      MCAPI_ENDP_ATTR_BUFFER_TYPE,         /* Buffer type, FIFO -
                                              Channel compatibility attribute
                                              */
      MCAPI_ENDP_ATTR_MEMORY_TYPE,         /* Shared/local (0-copy),
                                              blocking or non-blocking on
                                              limit - Channel compatibility
                                              attribute */
      MCAPI_ENDP_ATTR_NUM_PRIORITIES,      /* Number of priorities -
                                              Channel compatibility attribute
                                              */
      MCAPI_ENDP_ATTR_PRIORITY,            /* Priority on connected
                                              endpoint - Channel
                                              compatibility attribute */
      MCAPI_ENDP_ATTR_NUM_SEND_BUFFERS,    /* Number of send buffers at
                                              the current endpoint priority
                                              level */
      MCAPI_ENDP_ATTR_NUM_RECV_BUFFERS,    /* Number of available receive
                                              buffers */
      MCAPI_ENDP_ATTR_STATUS,              /* Endpoint status, connected,
                                              open etc. */
      MCAPI_ENDP_ATTR_TIMEOUT,             /* Timeout */
      MCAPI_ENDP_ATTR_END                  /* This should always be last */
};

/* MCAPI endpoint Attributes
 * Implementations may designate some or all attributes as read-only
 *
 */

/* MCAPI_ENDP_ATTR_MAX_PAYLOAD_SIZE */
typedef mcapi_int_t mcapi_endp_attr_max_payload_size_t;  /* Implementation
                                                            defined default
                                                            value */

/* MCAPI_ENDP_ATTR_BUFFER_TYPE */
typedef mcapi_uint_t mcapi_endp_attr_buffer_type_t;

/* Buffer Types */
enum mcapi_buffer_type {
      MCAPI_ENDP_ATTR_FIFO_BUFFER          /* Default */
};
```

```
/* MCAPI_ENDP_ATTR_MEMORY_TYPE */
typedef mcapi_uint_t mcapi_endp_attr_memory_type_t;

/*
 * The type refers to both the memory's locality, local, shared and remote.
 */

/* MCAPI Attribute Memory Locality Types */
enum mcapi_memory_type {
      MCAPI_ENDP_ATTR_LOCAL_MEMORY,       /* Zero copy operations not
                                             possible - Default */
      MCAPI_ENDP_ATTR_SHARED_MEMORY,      /* The user buffer provided
                                             by the sender is a shared
                                             memory buffer, zero copy
                                             possible */
      MCAPI_ENDP_ATTR_REMOTE_MEMORY
};

/* MCAPI_ENDP_ATTR_NUM_PRIORITIES */
typedef mcapi_int_t mcapi_endp_attr_num_priorities_t; /* Implementation
                                                         defined default
                                                         value */

/* MCAPI_ENDP_ATTR_PRIORITY */
typedef mcapi_uint_t mcapi_endp_attr_priority_t;      /* A lower number
                                                         means higher
                                                         priority. A value
                                                         of
                                                         MCAPI_MAX_PRORITY
                                                         (0) denotes the
                                                         highest priority
                                                         MCAPI_MAX_PRORITY -
                                                         Default */

/* MCAPI_ENDP_ATTR_NUM_SEND_BUFFERS */
typedef mcapi_int_t mcapi_endp_attr_num_send_buffers_t; /* Implementation
                                                           defined default
                                                           value */

/* MCAPI_ENDP_ATTR_NUM_RECV_BUFFERS */
typedef mcapi_uint_t mcapi_endp_attr_num_recv_buffers_t; /* Number of
                                                            receive
                                                            buffers
                                                            available,
                                                            can for example
                                                            be used for
                                                            throttling.
                                                            Implementation
                                                            defined default
                                                            value */

/* MCAPI_ENDP_ATTR_STATUS */
typedef mcapi_uint_t mcapi_endp_attr_status_t;

/*
 * MCAPI Endpoint Status Flags, are used to query the status of an endpoint,
 * e.g. if it is connected and if so what type of channel, direction, etc.
 *
 * Note: The lower 16 bits are defined in mcapi.h whereas the upper 16 bits
 * are reserved for implementation specific purposes and if used must be
 * defined in implementation_spec.h. It is therefore recommended that the
```

```
 * upper 16 bits are masked off at the application level.
 *
 *          0x00000000
 *                ----  mcapi.h
 *            ----            implementation_spec.h
 *
 * Default = 0x00000000
 *
 */
#define MCAPI_ENDP_ATTR_STATUS_CONNECTED      0x00000001 /* The endpoint
                                                            is connected */
#define MCAPI_ENDP_ATTR_STATUS_OPEN           0x00000002 /* The channel
                                                            is open */
#define MCAPI_ENDP_ATTR_STATUS_OPEN_PENDING   0x00000004 /* An open request
                                                            is pending */
#define MCAPI_ENDP_ATTR_STATUS_CLOSE_PENDING  0x00000008 /* An close
                                                            request is
                                                            pending */
#define MCAPI_ENDP_ATTR_STATUS_PKTCHAN        0x00000010 /* The channel
                                                            is a packet
                                                            channel */
#define MCAPI_ENDP_ATTR_STATUS_SCLCHAN        0x00000020 /* The channel
                                                            is a scalar
                                                            channel */
#define MCAPI_ENDP_ATTR_STATUS_SEND           0x00000040 /* Endpoint is
                                                            the send side
                                                            of the
                                                            channel */
#define MCAPI_ENDP_ATTR_STATUS_RECEIVE        0x00000080 /* Endpoint is
                                                            the receive
                                                            side of the
                                                            channel */

/* MCAPI_ENDP_ATTR_TIMEOUT */
typedef mcapi_timeout_t mcapi_endp_attr_timeout_t;/* Timeout for blocking
                                                     send and receive
                                                     functions Default =
                                                     MCAPI_TIMEOUT_INFINITE
                                                     and a value of
                                                     MCAPI_TIMEOUT_IMMEDIATE
                                                     (or 0) means that the
                                                     function will return
                                                     immediately, with
                                                     success or failure */

/*
 * MCAPI endpoint attribute #'s
 * The first range (64) is assigned to MCA
 * Organizations can apply to the MCA for endpoint attribute # ranges
 * One organization can be assigned multiple ranges.
 */
#define MCAPI_MAX_NUM_ENDP_ATTRS_MCA      64    /* MCAPI endpoint attributes
                                                   reserved for MCA */
#define MCAPI_MAX_NUM_ENDP_ATTRS_OTHER    32    /* MCAPI endpoint attributes
                                                   reserved for other
                                                   organizations */

/*
 * Macros for calculating starting and ending attribute numbers for an MCA
 * assigned attribute range.
 *
```

```
 * x = assigned range. One organization can be assigned multiple ranges.
 */
#define mcapi_start_endp_attribute(x)  MCAPI_MAX_NUM_ENDP_ATTRS_MCA + \
                                       (x * MCAPI_MAX_NUM_ENDP_ATTRS_OTHER)
#define mcapi_end_endp_attribute(x)    MCAPI_MAX_NUM_ENDP_ATTRS_MCA + \
                                       (x + 1) * \
                                       MCAPI_MAX_NUM_ENDP_ATTRS_OTHER

/*
 * MCAPI Initialization information
 * In addition to the MCAPI defined information implementations may
 * include implementation specific information.
 */
typedef struct
{
    mcapi_uint_t  mcapi_version;          /* MCAPI version, the three last
                                             (rightmost) hex digits are the
                                             minor number and those left
                                             of minor the major number */
    mcapi_uint_t  organization_id;        /* Implementation
                                             vendor/organization id */
    mcapi_uint_t  implementation_version; /* Implementation version, the
                                             format is implementation
                                             defined  */
    mcapi_uint_t  number_of_domains;      /* Number of domains in the
                                             topology */
    mcapi_uint_t  number_of_nodes;        /* Number of nodes in the domain,
                                             can be used for basic per
                                             domain topology discovery */
    mcapi_uint_t  number_of_ports;        /* Number of available ports on
                                             the local node */
    impl_info_t   *impl_info;             /* This structure has to be
                                             defined by the implementor in
                                             implementation_spec.h */
}mcapi_info_t;

/* MCAPI Organization specific reserved endpoint attribute numbers */
typedef struct
{
    mcapi_uint_t  organization_id;        /* Implementation
                                             vendor/organization id */
    mcapi_uint_t  start_endp_attr_num;    /* Starting MCA provided vendor
                                             specific attribute number */
    mcapi_uint_t  end_endp_attr_num;      /* Ending MCA provided vendor
                                             specific attribute number */
} mcapi__endp_attr_range_t;


/* In/out parameter indication macros */
#ifndef MCAPI_IN
#define MCAPI_IN const
#endif /* MCAPI_IN */

#ifndef MCAPI_OUT
#define MCAPI_OUT
#endif /* MCAPI_OUT */


/* Alignment macros */

#ifndef MCAPI_DECL_ALIGNED
```

```
#define MCAPI_DECL_ALIGNED MCA_DECL_ALIGNED
#endif

#ifndef MCAPI_BUF_ALIGN
#define MCAPI_BUF_ALIGN MCA_BUF_ALIGN
#endif

/*
 * Function prototypes
 */

/* Initialization, node and endpoint management */

void mcapi_initialize(
      MCAPI_IN mcapi_domain_t domain_id,
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_IN mcapi_node_attributes_t* mcapi_node_attributes,
      MCAPI_IN mcapi_param_t* init_parameters,
      MCAPI_OUT mcapi_info_t* mcapi_info,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_finalize(
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_domain_t mcapi_domain_id_get(
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_node_t mcapi_node_id_get(
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_node_init_attributes(
      MCAPI_OUT mcapi_node_attributes_t* mcapi_node_attributes,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_node_set_attribute(
      MCAPI_OUT mcapi_node_attributes_t* mcapi_node_attributes,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_IN void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* status
);

void mcapi_node_get_attribute(
      MCAPI_IN mcapi_domain_t domain_id,
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_OUT void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_endpoint_t mcapi_endpoint_create(
      MCAPI_IN mcapi_port_t port_id,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
```

```
void mcapi_endpoint_get_i(
      MCAPI_IN mcapi_domain_t domain_id,
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_IN mcapi_port_t port_id,
      MCAPI_OUT mcapi_endpoint_t* endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_endpoint_t mcapi_endpoint_get(
      MCAPI_IN mcapi_domain_t domain_id,
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_IN mcapi_port_t port_id,
      MCAPI_IN mcapi_timeout_t timeout,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_endpoint_delete(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_endpoint_get_attribute(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_OUT void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_endpoint_set_attribute(
      MCAPI_IN mcapi_endpoint_t endpoint,
      MCAPI_IN mcapi_uint_t attribute_num,
      MCAPI_OUT const void* attribute,
      MCAPI_IN size_t attribute_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Message functions */

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

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

```
void mcapi_msg_recv_i(
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_msg_recv(
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT void* buffer,
      MCAPI_IN size_t buffer_size,
      MCAPI_OUT size_t* received_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint_t mcapi_msg_available(
      MCAPI_IN mcapi_endpoint_t receive_endoint,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Packet channel functions */

void mcapi_pktchan_connect_i(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_recv_open_i(
      MCAPI_OUT mcapi_pktchan_recv_hndl_t* receive_handle,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_send_open_i(
      MCAPI_OUT mcapi_pktchan_send_hndl_t* send_handle,
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_send_i(
      MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
      MCAPI_IN void* buffer,
      MCAPI_IN size_t size,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

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

```
void mcapi_pktchan_recv_i(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT void** buffer,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_recv(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT void** buffer,
      MCAPI_OUT size_t* received_size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint_t mcapi_pktchan_available(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_release(
      MCAPI_IN void* buffer,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_boolean_t mcapi_pktchan_release_test(
      MCAPI_IN void* buffer,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_recv_close_i(
      MCAPI_IN mcapi_pktchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_pktchan_send_close_i(
      MCAPI_IN mcapi_pktchan_send_hndl_t send_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Scalar channel functions */

void mcapi_sclchan_connect_i(
      MCAPI_IN mcapi_endpoint_t send_endpoint,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_recv_open_i(
      MCAPI_OUT mcapi_sclchan_recv_hndl_t* receive_handle,
      MCAPI_IN mcapi_endpoint_t receive_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
```

```
void mcapi_sclchan_send_open_i(
      MCAPI_OUT mcapi_sclchan_send_hndl_t* send_handle,
      MCAPI_IN mcapi_endpoint_t  send_endpoint,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_send_uint64(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint64_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_send_uint32(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint32_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_send_uint16(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint16_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_send_uint8(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_IN mcapi_uint8_t dataword,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint64_t mcapi_sclchan_recv_uint64(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint32_t mcapi_sclchan_recv_uint32(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint16_t mcapi_sclchan_recv_uint16(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint8_t mcapi_sclchan_recv_uint8(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_uint_t mcapi_sclchan_available(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_sclchan_recv_close_i(
      MCAPI_IN mcapi_sclchan_recv_hndl_t receive_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);
```

```
void mcapi_sclchan_send_close_i(
      MCAPI_IN mcapi_sclchan_send_hndl_t send_handle,
      MCAPI_OUT mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);


/* Non-blocking management functions */

mcapi_boolean_t mcapi_test(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT size_t* size,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

mcapi_boolean_t mcapi_wait(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT size_t* size,
      MCAPI_IN mcapi_timeout_t timeout,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

unsigned int mcapi_wait_any(
      MCAPI_IN size_t number,
      MCAPI_IN mcapi_request_t* requests,
      MCAPI_OUT size_t* size,
      MCAPI_IN mcapi_timeout_t timeout,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

void mcapi_cancel(
      MCAPI_IN mcapi_request_t* request,
      MCAPI_OUT mcapi_status_t* mcapi_status
);

/* Support functions */
char* mcapi_display_status(
      MCAPI_IN mcapi_status_t mcapi_status,
      MCAPI_OUT char* status_message,
      MCAPI_IN size_t size
);


#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCAPI_H */
```

7.3 mca_impl_spec.h

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
```

```
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mca_impl_spec.h
 *
 * Version 2.015,  March 2011
 *
 * Note: THE TYPE DEFINTIONS AND TYPES IN THIS FILE ARE REQUIRED.
 * THE SPECIFICE TYPES AND VALUES ARE IMPLEMENTATION DEFINED AS DESCRIBED
 * BELOW.
 * THE TYPES AND VALUES BELOW ARE SPECIFIC TO Poly-Messenger/MCAPI AND
 * INCLUDED ONLY TO EXEMPLIFY
 *
*/

#ifndef MCA_IMPL_SPEC_H
#define MCA_IMPL_SPEC_H

#include "psi_mca_impl_spec.h"      /* PolyCore Software implementation
                                       specifics, INCLUDED ONLY TO
                                       EXEMPLIFY */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * MCA implementation specific type definitions.
 *
 * It is recommended that these types be either pointers or 32 bit
 * scalars, allowing simple arithmetic equality comparison (a == b).
 * Implementers may determine which of these type are used.
 *
 */
typedef unsigned int          mca_request_t;

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCA_IMPL_SPEC_H */
```

7.4 mcapi_impl_spec.h

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mca_impl_spec.h
 *
 * Version 2.015,  March 2011
 *
 * Note: THE TYPE DEFINTIONS AND TYPES IN THIS FILE ARE REQUIRED.
 * THE SPECIFICE TYPES AND VALUES ARE IMPLEMENTATION DEFINED AS DESCRIBED
 * BELOW.
 * THE TYPES AND VALUES BELOW ARE SPECIFIC TO Poly-Messenger/MCAPI AND
 * INCLUDED ONLY TO EXEMPLIFY
 *
*/

#ifndef MCAPI_IMPL_SPEC_H
#define MCAPI_IMPL_SPEC_H

#include "psi_mcapi_impl_spec.h"  /* PolyCore Software implementation
                                     specifics, INCLUDED ONLY TO EXEMPLIFY */

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * MCAPI implementation specific type definitions
 *
 * These types must be either pointers or 32 bit scalars, allowing simple
```

```
 * arithmetic equality  * comparison (a == b).
 * Implementers may determine which of these type are used.
 *
 */
typedef unsigned int mcapi_endpoint_t;    /* Note: The endpoint
                                             identifier must be
                                             topology unique. */
typedef unsigned int mcapi_pktchan_recv_hndl_t;
typedef unsigned int mcapi_pktchan_send_hndl_t;
typedef unsigned int mcapi_sclchan_send_hndl_t;
typedef unsigned int mcapi_sclchan_recv_hndl_t;

typedef struct {
      void* pMemory;
      unsigned nMemorySize;
      PMMCAPIParameters* pMCAPIParams;
      int   nContextType;
} mcapi_param_t;


/* Number of MCAPI reserved ports, starting at port 0. Reserved ports can be
 * used for implementation specific purposes.
 *
 */
#define MCAPI_NUM_RESERVED_PORTS                        1     /* Number of
reserved

         ports starting at

         port 0 */

/* Implementation defined MCAPI MIN and MAX values.
 *
 * Implementations may parameterize implementation specific max values,
 * smaller that the MCAPI max values. Implementations must specify what
 * those smaller values are and how they are set.
 *
 */
#define MCAPI_MAX_DOMAIN        (2 << 14) - 1  /* Maximum value for domain */
#define MCAPI_MAX_NODE           (2 << 7) - 1  /* Maximum value for node */
#define MCAPI_MAX_PORT           (2 << 7) - 1  /* Maximum value for port */
#define MCAPI_MAX_MESSAGE_SIZE  (2 << 31) - 1  /* Maximum message size */
#define MCAPI_MAX_PACKET_SIZE   (2 << 31) - 1  /* Maximum packet size */

/*
 * Implementations may parameterize implementation specific priority min
 * value  * and set the number of reserved ports. Implementations must
 * specify what  * those values are and how they are set.
 */
#define MCAPI_MIN_PORT MCAPI_NUM_RESERVED_PORTS /* Minimum value for port */
#define MCAPI_MIN_PRORITY       (2 << 31) - 1    /* Minimum priority value */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCAPI_IMPL_SPEC_H */
```

7.5 mcapi_v1000_to_v2000.h

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mcapi_v1000_to_v2000.h
 *
 * Version 2.015, March 2011
 *
 */

#ifndef MCAPI_V1000_TO_V2000_H_
#define MCAPI_V1000_TO_V2000_H_

#include "mcapi.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * Type definitions not used in MCAPI 2.015
 */
typedef unsigned int          mcapi_version_t;

#define MCAPI_INFINITE
#define MCAPI_ENDPOINT_INVALID
#define MCAPI_INFINITE MCAPI_TIMEOUT_INFINITE

/*
 * Max value conversions
 */
#define MCAPI_MAX_NUM_NODES      MCAPI_MAX_NODE + 1   /* Max number
```

```
                                                          of nodes */
#define MCAPI_MAX_NUM_ENDPOINTS  MCAPI_MAX_PORT       /* Max number of
                                                          endpoints per
                                                          node */
#define MCAPI_MAX_NUM_CHANNEL_HANDLES MCAPI_MAX_CHANNEL_HANDLE + 1 /* Max
                                                          number of channel
                                                          handles per
                                                          endpoint */
#define MCAPI_MAX_NUM_REQUESTS   MCAPI_MAX_REQUEST    /* Max number of
                                                          request handles
                                                          per endpoint */
#define MCAPI_MAX_NUM_PRORITIES  MCAPI_MAX_PRORITY    /* Max number of
                                                          priorities */

/*
 * Status code conversions
 */
#define MCAPI_INCOMPLETE            MCAPI_PENDING
#define MCAPI_EPARAM                MCAPI_ERR_PARAMETER
#define MCAPI_EDOMAIN_NOTVALID      MCAPI_ERR_DOMAIN_INVALID
#define MCAPI_ENODE_NOTVALID        MCAPI_ERR_NODE_INVALID
#define MCAPI_ENO_INIT              MCAPI_ERR_NODE_INITFAILED
#define MCAPI_INITIALIZED           MCAPI_ERR_NODE_INITIALIZED
#define MCAPI_ENODE_NOTINIT         MCAPI_ERR_NODE_NOTINIT
#define MCAPI_ENO_FINAL             MCAPI_ERR_NODE_FINALFAILED
#define MCAPI_EPORT_NOTVALID        MCAPI_ERR_PORT_INVALID
#define MCAPI_ENOT_ENDP             MCAPI_ERR_ENDP_INVALID
#define MCAPI_EENDP_NOTALLOWED      MCAPI_ERR_ENDP_NOPORTS
#define MCAPI_EENDP_LIMIT           MCAPI_ERR_ENDP_LIMIT
#define MCAPI_EENDP_ISCREATED       MCAPI_ERR_ENDP_EXISTS
#define MCAPI_ENOT_OWNER            MCAPI_ERR_ENDP_NOTOWNER
#define MCAPI_EENDP_REMOTE          MCAPI_ERR_ENDP_REMOTE
#define MCAPI_EATTR_INCOMP          MCAPI_ERR_ATTR_INCOMPATIBLE
#define MCAPI_EATTR_SIZE            MCAPI_ERR_ATTR_SIZE
#define MCAPI_EATTR_NUM             MCAPI_ERR_ATTR_NUM
#define MCAPI_EATTR_VALUE           MCAPI_ERR_ATTR_VALUE
#define MCAPI_EATTR_NOTSUPPORTED    MCAPI_ERR_ATTR_NOTSUPPORTED
#define MCAPI_EREAD_ONLY            MCAPI_ERR_ATTR_READONLY
#define MCAPI_EMESS_LIMIT           MCAPI_ERR_MSG_LIMIT
#define MCAPI_ETRUNCATED            MCAPI_ERR_MSG_TRUNCATED
#define MCAPI_ETRANSMISSION         MCAPI_ERR_TRANSMISSION
#define MCAPI_ENO_MEM               MCAPI_ERR_MEM_LIMIT
#define MCAPI_EREQ_TIMEOUT          MCAPI_TIMEOUT
#define MCAPI_ENO_REQUEST           MCAPI_ERR_REQUEST_LIMIT
#define MCAPI_EPRIO                 MCAPI_ERR_PRIORITY
#define MCAPI_ECHAN_OPEN            MCAPI_ERR_CHAN_OPEN
#define MCAPI_ECHAN_TYPE            MCAPI_ERR_CHAN_TYPE
#define MCAPI_ECONNECTED            MCAPI_ERR_CHAN_CONNECTED
#define MCAPI_ECHAN_OPENPENDING     MCAPI_ERR_CHAN_OPENPENDING
#define MCAPI_EDIR                  MCAPI_ERR_CHAN_DIRECTION
#define MCAPI_ENOT_OPEN             MCAPI_ERR_CHAN_NOTOPEN
#define MCAPI_ENOT_HANDLE           MCAPI_ERR_CHAN_INVALID
#define MCAPI_ENOTREQ_HANDLE        MCAPI_ERR_REQUEST_INVALID
#define MCAPI_EPACK_LIMIT           MCAPI_ERR_PKT_LIMIT
#define MCAPI_ENOT_VALID_BUF        MCAPI_ERR_BUF_INVALID
#define MCAPI_ESCL_SIZE             MCAPI_ERR_GENERAL
#define MCAPI_EREQ_CANCELED         MCAPI_ERR_REQUEST_CANCELLED
#define MCAPI_LAST_CODE             MCAPI_STATUSCODE_END

/*
 * Endpoint Atribute conversions
```

```
 */
#define mcapi_attr_endpoint_type        mcapi_endp_attr_type_t
#define mcapi_attr_num_priorities       mcapi_endp_attr_num_priorities_t
#define mcapi_attr_num_buffers          mcapi_endp_attr_num_send_buffers_t
#define mcapi_attr_message_buffer_size  mcapi_endp_attr_max_payload_size_t
#define mcapi_attr_buffer_type          mcapi_endp_attr_buffer_type_t
#define mcapi_attr_memory_type          mcapi_endp_attr_memory_type_t
#define mcapi_attr_timeout              mcapi_endp_attr_timeout_t
#define mcapi_attr_endp_priority        mcapi_endp_attr_priority_t
#define mcapi_attr_endp_status          mcapi_endp_attr_status_t
#define mcapi_attr_num_recv_buffers_available \
                                        mcapi_endp_attr_num_recv_buffers_t
#define MCAPI_ATTR_NUM_PRIORITIES       MCAPI_ENDP_ATTR_NUM_PRIORITIES
#define MCAPI_ATTR_NUM_BUFFERS          MCAPI_ENDP_ATTR_NUM_SEND_BUFFERS
#define MCAPI_ATTR_NUM_RECV_BUFFERS_AVAILABLE \
                                        MCAPI_ENDP_ATTR_NUM_RECV_BUFFERS
#define MCAPI_ATTR_ENDP_STATUS          MCAPI_ENDP_ATTR_ENDP_STATUS
#define MCAPI_ATTR_TIMEOUT              MCAPI_ENDP_ATTR_TIMEOUT
#define MCAPI_ATTR_BUFFER_SIZE          MCAPI_ENDP_ATTR_MAX_PAYLOAD_SIZE
#define MCAPI_ATTR_BUFFER_TYPE          MCAPI_ENDP_ATTR_BUFFER_TYPE
#define MCAPI_ATTR_MEMORY_TYPE          MCAPI_ENDP_ATTR_MEMORY_TYPE
#define MCAPI_ATTR_ENDP_PRIORITY        MCAPI_ENDP_ATTR_PRIORITY
#define MCAPI_ATTR_END                  MCAPI_ENDP_ATTR_END

/*
 * Endpoint Status Flag conversions
 */
#define MCAPI_CREATED               MCAPI_ENDP_CREATED
#define MCAPI_CONNECTED             MCAPI_ENDP_ATTR_STATUS_CONNECTED
#define MCAPI_OPEN                  MCAPI_ENDP_ATTR_STATUS_OPEN
#define MCAPI_OPEN_PENDING          MCAPI_ENDP_ATTR_STATUS_OPEN_PENDING
#define MCAPI_CLOSE_PENDING         MCAPI_ENDP_ATTR_STATUS_CLOSE_PENDING
#define MCAPI_PKT                   MCAPI_ENDP_ATTR_STATUS_PKTCHAN
#define MCAPI_SCL                   MCAPI_ENDP_ATTR_STATUS_SCLCHAN
#define MCAPI_SEND                  MCAPI_ENDP_ATTR_STATUS_SEND
#define MCAPI_RECEIVE               MCAPI_ENDP_ATTR_STATUS_RECEIVE
#define MCAPI_GET_PENDING           MCAPI_ENDP_ATTR_STATUS_GET_PENDING

/* Attribute number ranges */
#define MCAPI_MAX_NUM_ATTRS_MCA     MCAPI_MAX_NUM_ENDP_ATTRS_MCA
#define MCAPI_MAX_NUM_ATTRS_OTHER   MCAPI_MAX_NUM_ENDP_ATTRS_OTHER

/* MCA organization id's */
#define ORG_ID_PSI            MCA_ORG_ID_PSI
#define ORG_ID_FSL            MCA_ORG_ID_FSL
#define ORG_ID_TBA            MCA_ORG_ID_TBA


/*
 * Function prototype conversions
 */

/*
 *  Initialization, node and endpoint management
 */

extern void mcapi_1000_initialize(
            MCAPI_IN mcapi_node_t node_id,
            MCAPI_OUT mcapi_version_t* mcapi_version,
            MCAPI_OUT mcapi_status_t* mcapi_status);
```

```
#define mcapi_get_node_id(mcapi_status)   mcapi_node_id_get(mcapi_status)

#define mcapi_create_endpoint(port_id, mcapi_status) \
        mcapi_endpoint_create(port_id, mcapi_status)

#define mcapi_1000_endpoint_get_i(node_id, port_id, endpoint, \
        request, mcapi_status) mcapi_endpoint_get_i(0, node_id, \
        port_id, endpoint, request, mcapi_status)

#define mcapi_1000_endpoint_get(node_id, port_id, mcapi_status) \
        mcapi_endpoint_get(0, node_id, port_id, 0, mcapi_status)

#define mcapi_delete_endpoint(endpoint, mcapi_status) \
        mcapi_endpoint_delete(endpoint, mcapi_status)

#define mcapi_get_endpoint_attribute(endpoint, attribute_num, attribute, \
        attribute_size, mcapi_status) \
        mcapi_endpoint_get_attribute(endpoint, attribute_num, attribute, \
        attribute_size, mcapi_status)

#define mcapi_endpoint_set_attribute(endpoint, attribute_num, attribute, \
        attribute_size, mcapi_status) \
        mcapi_set_endpoint_attribute(endpoint, attribute_num, attribute, \
        attribute_size, mcapi_status)
```

```
/* Packet channel functions */

#define mcapi_connect_pktchan_i(send_endpoint, receive_endpoint, request, \
        mcapi_status) \
        mcapi_pktchan_connect_i(send_endpoint, receive_endpoint, request, \
        mcapi_status)

#define mcapi_open_pktchan_i(receive_handle, receive_endpoint, request, \
        mcapi_status) \
        mcapi_pktchan_recv_open_i(receive_handle, receive_endpoint, \
        request, mcapi_status)

#define mcapi_open_pktchan_send_i(send_handle, send_endpoint, request, \
        mcapi_status) \
        mcapi_pktchan_send_open_i(send_handle, send_endpoint, request, \
        mcapi_status)

#define mcapi_pktchan_free(buffer, mcapi_status) \
        mcapi_pktchan_release(buffer, mcapi_status)

/* Scalar channel functions */

#define mcapi_connect_sclchan_i(send_endpoint, receive_endpoint, \
        request, mcapi_status) \
        mcapi_sclchan_connect_i(send_endpoint, receive_endpoint, request, \
        mcapi_status)

#define mcapi_open_sclchan_recv_i(receive_handle, receive_endpoint, \
        request, mcapi_status) \
        mcapi_sclchan_recv_open_i(receive_handle, receive_endpoint, \
        request, mcapi_status)

#define mcapi_open_sclchan_send_i(send_handle, send_endpoint, request, \
        mcapi_status) \
        mcapi_sclchan_send_open_i(send_handle, send_endpoint, request, \
        mcapi_status)

/* Non-blocking management functions */

#define mcapi_1000_wait(request, size, mcapi_status, timeout) \
        mcapi_wait(request, size, timeout, mcapi_status)

#define mcapi_1000_wait_any(number, requests, size, mcapi_status, timeout) \
        mcapi_wait_any(number, *requests, size, timeout, mcapi_status)

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCAPI_V1000_TO_V2000_H_ */
```

7.6 mcapi_1000_functions.c

```
/*
 * Copyright (c) 2011, The Multicore Association All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * (1) Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *
 * (2) Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 *
 * (3) Neither the name of the Multicore Association nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 *     THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *     "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *     LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *     COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *     INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *     BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *     LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *     CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *     LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *     ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *     POSSIBILITY OF SUCH DAMAGE.
 *
 * mcapi_1000_functions.c
 *
 * Version 2.015, March 2011
 *
 */

#include "mcapi.h"

void mcapi_1000_initialize(
      MCAPI_IN mcapi_node_t node_id,
      MCAPI_OUT mcapi_version_t* mcapi_version,
      MCAPI_OUT mcapi_status_t* mcapi_status){

      mcapi_version_t   version = 0;
      mcapi_info_t info;

      mcapi_initialize(0, node_id, MCAPI_NULL, MCAPI_NULL, &mcapi_info, \
                  mcapi_status);
      *mcapi_version = mcapi_info->mcapi_version;
}
```

8. Appendix C: MCAPI Specification License Agreement

PLEASE READ THIS MULTICORE ASSOCIATION LICENSE AGREEMENT ("LICENSE" or "AGREEMENT") CAREFULLY BEFORE DOWNLOADING THE MULTICORE ASSOCIATION'S MULTICORE COMMUNICATIONS API (MCAPI) SPECIFICATION AND/OR THE MULTICORE ASSOCIATION'S RESOURCE MANAGEMENT API (MRAPI) SPECIFICATION. BY USING THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, DO NOT DOWNLOAD OR USE THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION. FOR PURPOSES OF THIS LICENSE, "YOU" MEANS THE INDIVIDUAL OR ENTITY THAT IS BEING LICENSED TO USE THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION UNDER THE TERMS AND CONDITIONS OF THIS LICENSE. IF AN ENTITY IS BEING LICENSED, THEN THE PERSON AGREEING TO BE BOUND REPRESENTS THAT THE ENTITY HAS AUTHORIZED AND DIRECTED THE MAKING OF THIS LICENSE.

1. General. The specification and/or documentation accompanying this License whether on disk, in read only memory, on any other media or in any other form (collectively the "MCAPI Specification and/or MRAPI Specification") are licensed, not sold, to you by the Multicore Association for use only under the terms of this License, and Multicore Association reserves all rights not expressly granted to you. The rights granted herein are limited in scope per the terms of this License. The terms of this License will govern any upgrades provided by Multicore Association that replace and/or supplement the original Multicore Association MCAPI Specification and/or MRAPI Specification, unless such upgrade is accompanied by a separate license in which case the terms of that license will govern.

2. Permitted License Uses and Restrictions. This License allows you to download and use the Multicore Association MCAPI Specification and/or MRAPI Specification only as expressly permitted under this License. You are permitted to use the MCAPI Specification and/or MRAPI Specification for your own business purposes, to design, create, install, test, utilize, and distribute to other companies or individuals derived works that you will own and for which you will be solely responsible. As indicated in the MCAPI Specification and/or MRAPI Specification, you may document implementation specific behavior in the sections designated in the MCAPI Specification and/or MRAPI Specification; however, such documentation shall not constitute a modification of the MCAPI Specification and/or MRAPI Specification and must not be presented as such. You may create hard copies of the MCAPI Specification and/or MRAPI Specification for your own use and you may view electronic versions of the MCAPI Specification and/or MRAPI Specification on your personal computer (or other electronic viewing device) for your own use. If you have employees, you may authorize your employees to use the MCAPI Specification and/or MRAPI Specification in accordance with the provisions of this License and to make or view copies of the MCAPI Specification and/or MRAPI Specification in accordance with the foregoing provisions of this Section 2. Any use or copying of the MCAPI Specification and/or MRAPI Specification that is not specifically permitted by this License shall constitute a violation of this License and a default by you.

3. Enhancements by Multicore Association. In the event that at any time Multicore Association makes any enhancement, update, or modification to the MCAPI Specification and/or MRAPI Specification or becomes the owner of any new enhancement, update, or modification to the MCAPI Specification and/or MRAPI Specification, then upon notice to you, you shall have the same right and license to use and exploit the same as it is granted hereunder with respect to the original MCAPI Specification and/or MRAPI Specification. You agree that all rights in and to enhancements, updates, and modifications effected by Multicore Association, if any, to the MCAPI Specification and/or MRAPI Specification, shall remain the sole and exclusive property of Multicore Association. Nothing contained herein shall obligate Multicore Association to either create or make available any new enhancement, update or modifications to the Multicore Association MCAPI Specification and/or MRAPI Specification. Multicore Association has no obligation to provide any maintenance and technical support services.

4. Enhancements by You. You shall have no right to independently modify, improve, or enhance the Multicore Association MCAPI Specification and/or MRAPI Specification. However, your derived work may support additional behavior that is not defined in the specification as long as that behavior doesn't prevent your derived work from correctly implementing the MCAPI Specification and/or MRAPI Specification.

5. Use of Trademark. You agree that you will not, without the prior written consent of Multicore Association, use in advertising, publicity, packaging, labeling, or otherwise any trade name, trademark, service mark, symbol, or any other identification owned by Multicore Association to identify any of its products or services.

6. Delivery. The Multicore Association MCAPI Specification and/or MRAPI Specification will be available for downloading on Multicore Association's website in accordance with the current policies and procedures of Multicore Association. Multicore Association reserves the right to modify such procedures in its sole and absolute discretion.

7. Term and Termination.

a. Term. The term of this Agreement shall continue so long as you are not in Default as set forth in Section 7(b).

b. Termination. Multicore Association shall have the right to terminate this Agreement and the License granted herein if you commit an act of or are subject to any Default. A Default means you breach a term or condition of this Agreement or you assign or purport to assign any of the rights granted herein without the prior written approval of Multicore Association. Upon the occurrence of a Default, this Agreement shall immediately terminate. Multicore Association's rights as set forth in this Section 7(b) are cumulative and, except as provided herein, are in addition to any other rights Multicore Association may have at law or in equity.

c. Effect of Termination. Upon the expiration or termination of this Agreement, the rights granted to you hereunder shall immediately cease and discontinue, and you shall be required to immediately return any and all materials and deliverables provided to you under this Agreement, including without limitation, the Multicore Association MCAPI Specification and/or MRAPI Specification. The provisions contained in Sections 1,4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17 of this Agreement shall survive any such termination or expiration.

8. No Warranty. Multicore Association provides no warranty for the Multicore Association MCAPI Specification and/or MRAPI Specification.

9. Disclaimer of Warranties. YOU EXPRESSLY ACKNOWLEDGE AND AGREE THAT USE OF THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION IS AT YOUR SOLE RISK AND THAT THE ENTIRE RISK AS TO SATISFACTORY QUALITY, PERFORMANCE, ACCURACY AND EFFORT IS WITH YOU. THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION IS PROVIDED "AS IS", WITH ALL FAULTS AND WITHOUT WARRANTY OF ANY KIND, AND MULTICORE ASSOCIATION HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH RESPECT TO THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES AND/OR CONDITIONS OF MERCHANTABILITY, OF SATISFACTORY QUALITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF ACCURACY, OF QUIET ENJOYMENT, AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. MULTICORE ASSOCIATION DOES NOT WARRANT AGAINST INTERFERENCE WITH YOUR ENJOYMENT OF THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION, THAT THE FUNCTIONS CONTAINED IN THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION WILL MEET YOUR REQUIREMENTS, THAT THE OPERATION OF THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION WILL BE CORRECTED. NO ORAL OR WRITTEN INFORMATION OR ADVICE GIVEN BY MULTICORE ASSOCIATION OR A MULTICORE ASSOCIATION AUTHORIZED REPRESENTATIVE SHALL

CREATE A WARRANTY. SHOULD THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION PROVE DEFECTIVE, YOU ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON APPLICABLE STATUTORY RIGHTS OF A CONSUMER, SO THE ABOVE EXCLUSION AND LIMITATIONS MAY NOT APPLY TO YOU.

10. POTENTIAL MISUSE OF MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION. YOU HEREBY ACKNOWLEDGE AND REPRESENT THAT YOU HAVE BEEN EXPRESSLY WARNED BY MULTICORE ASSOCIATION THAT THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION MAY BE INCOMPATIBLE WITH ANY APPLICATION OR END-USER PRODUCT, AND THAT SUCH MISUSE OF THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION COULD RESULT IN SIGNIFICANT PROPERTY DAMAGE AND/OR BODILY HARM. YOU FURTHER HEREBY ACKNOWLEDGE AND REPRSENT THAT YOU HAVE BEEN EXPRESSLY WARNED BY MULTICORE ASSOCIATION THAT THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION HAS NOT BEEN SPECIFICALLY DESIGNED OR TESTED FOR USE IN CONNECTION WITH ANY PARTICULAR APPLICATION, END-USER PRODUCT OR ACTIVITY. FOR EXAMPLE, BUT WITHOUT LIMITATION, YOU ACKNOWLEDGE THAT THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION HAS NOT BEEN SPECIFICALLY DESIGNED OR TESTED FOR USE IN CONNECTION WITH HIGH RISK ACTIVITIES SUCH AS THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL SYSTEMS, LIFE SUPPORT MACHINES OR OTHER EQUIPMENT IN WHICH THE FAILURE OF THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION OR THE DERIVED WORK YOU CREATE COULD LEAD TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE. ACCORDINGLY, YOU AGREE THAT YOUR USE OF THE MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION IN CONNECTION WITH ANY SUCH HIGH RISK ACTIVITES OR OTHERWISE SHALL BE AT YOUR SOLE DISCRETION, RISK AND RESPONSIBILITY.

11. Limitation of Liability. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT SHALL MULTICORE ASSOCIATION BE LIABLE FOR PERSONAL INJURY, OR ANY INCIDENTAL, SPECIAL, INDIRECT, CONSEQUENTIAL OR EXEMPLARY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, LOSS OF DATA, BUSINESS INTERRUPTION OR ANY OTHER COMMERCIAL DAMAGES OR LOSSES, ARISING OUT OF OR RELATED TO YOUR USE OR INABILITY TO USE THE MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION, HOWEVER CAUSED, REGARDLESS OF THE THEORY OF LIABILITY (CONTRACT, TORT OR OTHERWISE) AND EVEN IF MULTICORE ASSOCIATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF LIABILITY FOR PERSONAL INJURY, OR OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION MAY NOT APPLY TO YOU. In no event shall MULTICORE ASSOCIATION'S total liability to you for all damages (other than as may be required by applicable law in cases involving personal injury) exceed the amount of fifty dollars ($50.00). The foregoing limitations will apply even if the above stated remedy fails of its essential purpose.

12. Sole Responsibility for Use; Indemnification. You agree to be solely responsible for your use of the MCAPI Specification and/or MRAPI Specification including, but not limited to, use in designing, creating, installing, testing, utilizing, and distributing derived work. The entire risk as to the use, results and performance of the MCAPI Specification and/or MRAPI Specification is assumed by you. You agree that Multicore Association shall have no liability whatsoever for any use you make of the MCAPI Specification and/or MRAPI Specification. Upon a request by Multicore Association, you agree to indemnify, defend, reimburse, and hold harmless Multicore Association, and its members, and their officers, directors, shareholders, agents, employees, representatives, successors, and assigns, from and against any and all claims, demands, liabilities, damages, costs, expenses, actions or causes of action of any name or nature (including attorneys ' fees and costs), which may be asserted, claimed, prosecuted or established against them, or any of them, arising out of, in connection with or in any manner related to any or all of the following: (a) your use of the MCAPI Specification and/or MRAPI Specification; (b) any use, inability to use or malfunction of any derived work that you design or create from the MCAPI Specification and/or MRAPI Specification; and (c) your breach of or default under any provision of this License.

13. Export Law Assurances. You may not use or otherwise export or re-export the MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION except as authorized by United States law and the laws of the jurisdiction in which the MULTICORE ASSOCIATION MCAPI Specification and/or MRAPI Specification was obtained. In particular, but without limitation, the MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION may not be exported or re-exported (a) into (or to a national or resident of) any U.S. embargoed countries (currently Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria), or (b) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals or the U.S. Department of Commerce Denied Person's List or Entity List. By using the MULTICORE ASSOCIATION MCAPI SPECIFICATION AND/OR MRAPI SPECIFICATION, you represent and warrant that you are not located in, under control of, or a national or resident of any such country or on any such list.

14. Relationship of Parties. Neither this Agreement, nor any terms and conditions contained herein, may be construed as creating or constituting a partnership, joint venture, or agency relationship between the parties. Neither party will have the power to bind the other or incur obligations on the other party's behalf without the other party's prior written consent.

15. Waiver. No failure of either party to exercise or enforce any of its rights under this Agreement will act as a waiver of such rights.

16. Controlling Law and Severability; Location for Resolving Disputes. This License will be governed by and construed in accordance with the laws of the State of California, as applied to agreements entered into and to be performed entirely within California between California residents and without reference to its conflict of laws principles. This License shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded. If for any reason a court of competent jurisdiction finds any provision, or portion thereof, to be unenforceable, the remainder of this License shall continue in full force and effect. You agree that the exclusive jurisdiction for any claim or dispute with Multicore Association under this Agreement or relating in any way to your use of the MCAPI Specification and/or MRAPI Specification resides in the federal or state courts located in the State of California and you further agree and expressly consent to the exercise of personal jurisdiction in such courts in connection with any such dispute.

17. Complete Agreement; Governing Language. This License constitutes the entire agreement between the parties with respect to the use of the MULTICORE ASSOCIATION MCAPI Specification and/or MRAPI Specification licensed hereunder and supersedes all prior or contemporaneous understandings regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by MULTICORE ASSOCIATION. Any translation of this License is done for local requirements and in the event of a dispute between the English and any non-English versions, the English version of this License shall govern.