Multicore Resource API Specification V0.9.3

Document ID: MRAPI API Specification V0.9.3
Document Version: 0.9.3
Status: External Review
Distribution: MCA Members and External Reviewers

This page intentionally blank

Table of Contents

1 History

Multicore programming shares many concepts with parallel and distributed computing. Multiple computing elements interact to accomplish a given task. In order to implement this programmers need basic capabilities for synchronizing the various threads of computation and coordinating accesses to resources. These problems have been solved for traditional distributed systems using various forms of middleware, and for multicore desktops and servers by facilities in Symmetric Multiprocessing (SMP) enabled operating systems.

As multicore computing extends into embedded domains many aspects of computing heterogeneity emerge. This limits the ability of programmers to utilize middleware designed for distributed systems, or to rely on an SMP operating system. These forms of heterogeneity include memory architectures, instruction sets, general purpose cores, special purpose cores (or hardware acceleration), and even operating systems. Yet multicore programmers still face the same programming challenges. Semantically there is little difference between this computing context and the distributed or SMP context. While it could be argued that existing standards for resource management would suffice in the embedded context if re-implemented, two more concerns serve as barriers to this approach: (1) the requirements of distributed systems and SMP systems demand overheads of footprint and execution times that are unnecessary in closely-coupled and reliable embedded systems, and (2) embedded systems have some significant additional requirements not encompassed by existing standards.

The Multicore Resource API (MRAPI) is designed to address these issues by embracing the proven features of existing standards, while explicitly supporting the heterogeneous embedded multicore computing context (including combinations of hardware or software heterogeneity, for example different kinds of cores and accelerators, or different operating systems). This API was developed under the guidance of the Multicore Association (MCA) with participation by many of the MCA member companies. Furthermore, the MRAPI specification is intentionally scoped to fit within the roadmap defined by the MCA. The first component of that roadmap was the Multicore Communications API (MCAPI). MRAPI and MCAPI share many concepts, constructs, and goals.

2 Introduction

This document is intended to assist software developers who are either implementing MRAPI or writing applications that use MRAPI. The MRAPI goals are presented, followed by a brief review of existing standards to motivate the need for a new standard and a short discussion of how MRAPI is intended to provide the desired functionality while meeting the goals of the MRAPI working group.

2.1 MRAPI Goals

MRAPI is intended to provide essential capabilities for allowing applications to cooperatively manage shared resources in Multicore systems. Therefore, MRAPI runtimes are not required to provide secure enforcement of sharing policies, and furthermore MRAPI intentionally stops short of directly providing a full-featured dynamic resource manager capable of orchestrating a set of resources to satisfy constraints on performance, power, and quality of service. It is envisioned that MRAPI (in conjunction with other Multicore Association APIs) can serve as a valuable tool for implementing applications, as well as for implementing such full-featured resource managers and other types of layered services. For these reasons, the following set of goals were established and used to carefully weigh each feature included in the MRAPI API:

- Small application layer API, suitable for cores on a chip and chips on a board
- Easy to learn and use, incorporates essential feature set
- Supports lightweight and performant implementations
- Does not prevent/preclude use of complementary approaches
- Allows silicon providers to optimize their hardware and take advantage of hardware features
- Allows implementers to differentiate their offerings
- Can run on top of an OS, hypervisor, or bare metal
- Can co-exist with hardware acceleration
- Supports hardware implementations of the API
- Does not require homogeneous cores, operating system, or memory architecture on chip
- Provides source code level portability

2.2 Existing Standards and APIs

This section provides brief reviews of related standards and discusses how the MRAPI working group chose to address specific areas of functionality.

2.2.1 POSIX® Shared Memory

Shared memory is used to allow access to the same data by multiple threads of execution, which may be on the same processor or on multiple processors, thereby avoiding copying of the data. Posix provides a standard API for using shared memory, including allocation, deletion, mapping and managing the shared memory. Posix shared memory generally provides this functionality within the scope of one operating system, across one or multiple processor cores. This functionality is considered essential for Multicore programming, but is only one feature that MRAPI is intended to provide. The MRAPI working group wanted to add two additional features to a shared memory API, namely: (1) the ability for programmers to specify attributes of the memory to be shared (for example on-chip SRAM versus off-chip DDR), and (2) the ability for programmers to specify which elements of a Multicore system would be seeking access to the shared memory segment such that MRAPI could support shared memory for parts of Multicore systems where physical shared memory is non-uniformly accessible.

2.2.2 POSIX® Mutexes and Semaphores

Given the goals of MRAPI, the MRAPI working group considered POSIX® mutexes and semaphores (IEEE Standard 1003.1b) as having relevant functionality. However, the working group determined that condition variables and signaling should be considered within the scope of the future Multicore Association Multicore Task Management API (MTAPI) working group rather than the MRAPI working group. The rationale for this decision is that in order to properly implement condition variables and signaling one would require the ability to manage threads or processes, and this is what MTAPI will provide. Therefore the functionality should be considered on the Multicore Association roadmap, but deferred until MTAPI becomes available.

The POSIX® standard provides two forms of semaphores: mutexes (binary semaphores), and semaphores (counting semaphores).

2.2.2.1 POSIX® Mutexes

POSIX® mutexes are declared as part of the POSIX® Threads (pthreads) package. POSIX® mutexes are only guaranteed to work within a single process. It is possible on some systems to declare mutexes as global by setting the 'process-shared' attribute on the mutex, but implementations are not required to support this.

The following mutex types are defined within the POSIX® standard:

- PTHREAD_MUTEX_NORMAL - This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

- PTHREAD_MUTEX_ERRORCHECK - This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it shall return with an error. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

- PTHREAD_MUTEX_RECURSIVE - A thread attempting to relock this mutex without first unlocking it shall succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

- PTHREAD_MUTEX_DEFAULT - Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type which is not locked results in undefined behavior. An implementation may map this mutex to one of the other mutex types.

2.2.2.2 Mutex Analysis

After reviewing the pthreads API and semantics, the working group came to the following conclusions:

- POSIX® mutexes cannot always be shared between processes, it depends on the implementation
- forking a process that has POSIX® mutexes has pitfalls when mutexes are process-shared; for example the new child could inherit held locks from threads in the parent that do not exist in the child because fork always creates a child with one thread under the standard

- it is normally recommended that System V or POSIX®.1b semaphores should be used for process-to-process synchronization rather than pthreads mutexes (but this requires an SMP operating system at this time for Multicore)
- mutexes are useful for managing access to a single resource and simpler to use than System V and POSIX® semaphores
- priorities and associated protocols (PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT) are probably not something that could be guaranteed by MRAPI unless/until MTAPI is created; the MRAPI group chose to defer considering this feature of the POSIX®; we recognize that this puts the burden of dealing with priority inversion squarely on the applications programmer at this time
- the attribute 'types' for error checking are powerful and useful and are included in MRAPI

It was decided for MRAPI to cover a subset of POSIX® mutex functionality along with the following new requirements as follows:

- functionality equivalent to: `pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock`
- mutex attributes for reporting basic deadlock detection
- the ability to manage mutex attributes in a way that is consistent with MCAPI
- non-blocking operations in a way that is consistent with MCAPI
- default is for the mutex to be visible across processes/tasks/etc.
- does not require shared memory or SMP OS
- priority inversion cannot be dealt with by MRAPI until MTAPI comes along

2.2.2.3 POSIX® Semaphores

POSIX® semaphores are declared as part of either the 'Realtime' services or the 'XSI Interprocess Communications' services. XSI is the 'X/Open System Interface Extension', which is an extension to IEEE 1003.1b. The XSI interfaces are essentially the same as the System V IPC interfaces, which have been widely supported across most Unix systems. Functionality marked XSI is also an extension to the ISO C standard. Semaphores themselves are a POSIX® option and are not required on all implementations.

2.2.2.4 POSIX® Semaphores Analysis

After reviewing the semaphores API and semantics, the working group came to the following conclusions:

- According to the standard REALTIME semaphores may be process-private or process-shared; there is a lot of evidence that not all operating systems support process-shared REALTIME semaphores - notably Linux - and the standard does not state that process-shared is required.
- REALTIME supports named and unnamed semaphores. Named and unnamed semaphores have distinct operations, for example you must call `sem_close` to close a named semaphore and `sem_unlink` to destroy a named semaphore, while `sem_destroy` is used to close and destroy an un-named semaphore
- It is unspecified whether XSI functions can interoperate with the realtime interprocess communication facilities defined in Realtime.
- The REALTIME API is much simpler to use, the XSI interface is more tied to the operating system although it is clearly defined to be flexible and fast. REALTIME works on a single `sem_t` identifier, while XSI uses arrays of semaphores and arrays of operations per API call
- The `sem_wait, sem_trywait, and sem_timedwait` functions do provide simple deadlock detection errors

For MRAPI, the working group decided the following:

- Keep only the concept of named semaphores and also match semantics of MCAPI for endpoints
- Ignore the XSI type interface (avoid requiring the API user to create and manage a set of semaphores and semaphore operations per call)
- Provide non-blocking operations in a way that is consistent with MCAPI
- default is visible across processes/tasks/etc.
- does not require shared memory or SMP OS
- priority inversion cannot be dealt with by MRAPI until MTAPI comes along
- provide primitive deadlock reporting

2.2.3 Performance API (PAPI)

PAPI is a high performance oriented API that defines a common set of useful performance counters. PAPI provides a high level interface to start, stop, read, and register callbacks for counter overflow. PAPI provides metadata about resources of a system, including resources such as number of cores, number of counters, and shared libraries in use by an application.

PAPI also provides derived counters, such as IPC (Instructions Per Cycle), and timing and measurement functions, such as wall clock time consumed. It also provides mutexes, supports external monitoring of counters associated with a process or thread, and management functions concerning registering threads. PAPI has no memory management, has no concept of system partitioning, and the metadata is limited with respect to the total resources in an SOC.

The MRAPI working group views the PAPI features for metadata and performance counters as being a useful concept for the types of systems targeted by MRAPI.

2.2.4 IBM DaCS

IBM's DaCS library provides a portable API for managing distributed memory systems. It allows programmers to take advantage of the Cell processor's "Synergistic Processing Unit" (SPU) DMA engines, while still being able to execute the program on machines that do not have DMA engines. It provides functions for creating memory regions, registering memory regions on multiple distributed processors, and copying data in and out of those memory regions via DMA.

The MRAPI API shares several concepts with DaCS. In particular, both APIs provide functions for creating memory regions, registering them with multiple processors, and performing DMA operations between distributed shared memory and local memory. In order to minimize API complexity, MRAPI does not provide some features included in DaCS. In particular, the MRAPI APIs avoid the need to specify permissions on memory regions and limit DMA operations to linear or strided data arrays.

2.2.5 GASNet specification

The GASNet (Global-Address Space Networking) specification is an API aimed at implementers of Global-Address Space languages such as Unified Parallel C, and Titanium. Unlike MRAPI, GASNet is geared towards single program multiple data (SPMD) high-performance computing applications, rather than embedded systems.

GASNet is divided into a small core API, and a richer extended API. The core API consists of functions for job control, message passing (based on *Active Messages*), and atomicity control. The extended API enriches this functionality with memory-to-memory data transfer functions, lower-level register-to-memory operations, barrier synchronization, and threading support. The extended API has been designed to be implementable using only the core API, and the GASNet designers provide a portable reference implementation of the extended API in terms of the core API. However, high-performance GASNet implementations are expected to efficiently implement as much of the extended API as possible, exploiting platform-specific characteristics.

The GASNet memory-to-memory data transfer functionality shares similarities with remote memory operations in MRAPI. Unlike MRAPI, GASNet does not support scatter/gather operations. On the other hand, GASNet provides more sophisticated synchronization primitives for non-blocking operations, and supports register-to-memory copies. The extended GASNet API includes barrier synchronization, which is out of scope for MRAPI (as discussed in 2.2.2, co-ordination between processes is part of the scope of MTAPI). Another significant distinction is that GASNet provides for both message passing and remote memory operations. Message passing is *not* part of MRAPI, which is intended to co-exist with a message passing API such as MCAPI.

2.2.6 ARMCI library

ARMCI (Aggregate Remote Memory Copy Interface) is a library for remote memory access operations. ARMCI has been designed to be general purpose and portable, but is aimed at library implementers rather than application developers.

ARMCI shares similarities with remote memory operations in MRAPI. Unlike MRAPI, ARMCI provides guarantees on the order of remote memory operations issued by a given process. ARMCI uses *generalized I/O vectors* to support movement of multiple data segments between arbitrary remote and local memory locations. This is more general than the form of remote memory operations supported by MRAPI; the structure of MRAPI operations matches the ARMCI *strided* format, a special class of generalized I/O vectors where local and remote memory regions are regularly spaced. As well as put and get operations, ARMCI supports remote accumulate. This functionality is mainly useful in the high-performance/scientific computing domain (accumulation is also featured in the MPI-2 one-sided communication API). Accumulate operations are not present in MRAPI, which is not specifically geared towards this application domain.

2.3 The MRAPI Feature Set

MRAPI provides features in three categories: *Synchronization Primitives*, *Memory Primitives*, and *Metadata Primitives.*

The Synchronization Primitives (Section 4.3) are:

- **Mutexes** - binary primitives which could be provided by shared memory, a distributed runtime, or other means
- **Semaphores** - counting primitives that provide more capability than mutexes, although at perhaps a slight performance penalty
- **Reader/Writer Locks** – more advanced primitives that give the ability to support multiple readers concurrently while allowing only a single writer

The Memory Primitives (Section 4.4) are:

- **Shared Memory** - allows an application to allocate and manage shared memory regions where there is physical shared memory to support it, including special features which provide support for requesting memory with specific attributes, and support for allocation based on a set of sharing entities
- **Remote Memory** - allows an application to manage buffers that are shared but not implemented on top of physical shared memory; transport may be via chip-specific methods such as DMA transfers, SRIO, software cache, etc. Remote Memory Primitives also provide random access, scatter/gather, and hooks for software managed coherency.

The Metadata Primitives (Section 4.6) provide access to hardware information. They are not intended to be a facility for an application to create and manage its own metadata. This additional functionality could be a layered service or a future extension.

3 MRAPI Overview

The major MRAPI concepts are covered in the following sections. The concepts and supporting datatypes are defined to meet the goals stated in Section 2.1, including source code portability.

3.1 Definitions

timely - operation will return without having to block on any IPC to any remote nodes.
IPC – Inter-processor communication.
MCAPI – Multicore Communications API: communication standard defined by Multicore Association.
MRAPI – Multicore Resource API: resource management standard defined by Multicore Association in this document.
MTAPI – Multicore Task API: task management standard to be defined by Multicore Association.

3.2 MRAPI Domain

An MRAPI system is composed of one or more MRAPI *domains*. An MRAPI domain is a unique system global entity. Each MRAPI domain comprises a set of MRAPI *nodes* (Section 3.3). An MRAPI node may only belong to one MRAPI domain, while an MRAPI domain may contain one or more MRAPI nodes. The concept of a domain is shared amongst Multicore Association APIs and must be consistent (i) within any implementation that supports multiple APIs, and (ii) across implementations that require interoperability.

3.3 MRAPI Nodes

An MRAPI node is a process, a thread, or a processor. In other words, an MRAPI node is an independent thread of control. The working group explicitly chose to leave the definition of a node flexible in this regard because it allows applications to be written in the most portable fashion that can be allowed by underlying embedded SoC multicore hardware, while at the same time supporting more general-purpose multicore and manycore devices. Therefore, it is the decision of the working group that overly-specifying the definition of a node would unnecessarily restrict the degrees of freedom that are allowed for mapping functionality to the resources on an embedded multicore SoC. Such systems may include processor cores, hardware accelerators, and various virtualization technologies. The MRAPI working group believes that this definition of a node allows portability of software to be maintained at the interface level (e.g., the functional interface between nodes). However, the software implementation of a particular node cannot (and often should not) necessarily be preserved across a multicore SoC product line (or across product lines from different silicon providers) because a given node's functionality may be provided in different ways -- depending on the chosen multicore SoC.

The `mrapi_initialize()` call takes node number and domain number arguments, and an MRAPI application may only call `mrapi_initialize()` once per node. It is an error to call `mrapi_initialize()` multiple times from a given thread of control unless `mrapi_finalize()` is called between such calls. A given MRAPI implementation will specify what is a node (i.e., what thread of control - process, thread, or other -- is a node) for that implementation. A thread and process are just two examples of threads of control, and there could be others.

The concept of nodes in MRAPI is shared with other Multicore Association (MCA) 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.

3.4 MRAPI Synchronization Primitives

The MRAPI synchronization primitives include *mutexes*, *semaphores*, and *reader/writer locks*.

Mutexes are intended to be simple binary semaphores for exclusive locks. Semaphores allow for counting locks. The reader/writer locks can be used to implement shared (reader) and exclusive (writer) locking. Mutexes are intended to support very fast, close to the hardware implementations, while semaphores and reader/writer locks provide more flexibility to the application programmer at the expense of some performance.

MRAPI provides blocking and non-blocking functions for obtaining locks on the synchronization primitives. The non-blocking functions should be used in conjunction with the `mrapi_test()`, `mrapi_wait()` and `mrapi_wait_any()` functions to determine when the request has completed. There is no need for non-blocking semantics for unlock because it always happens immediately.

All of the synchronization primitives are supported across MRAPI domains by default but this may have a performance impact (e.g., chip-to-chip will necessarily be slower). Sharing across domains can be disabled by setting the `MRAPI_DOMAIN_SHARED` attribute of a synchronization primitive to `MRAPI_FALSE` (default is `MRAPI_TRUE`).

3.4.1 MRAPI Mutexes

The basic semantics of MRAPI mutexes can be summarized as follows: MRAPI mutexes are binary, they support recursion (but that is not the default), and they are intended to be the closest match to underlying HW acceleration in many systems. Recursive locking is allowed if the locking node already owns the lock, and if the mutex attributes have been set up to allow recursion. Recursive locking means that once a mutex is locked, it can be locked again by the lock owner before unlock is called. For each lock, a unique lock key is returned. This lock key must be provided when the mutex is unlocked. The implementation uses the keys to match the order of the lock/unlock calls. Recursive locking is disabled by default. Note that individual mutex attributes may vary, but they must be set before mutex creation and cannot be altered later.

3.4.2 MRAPI Semaphores

Semaphores are differentiated from mutexes in that they support counting locks. Therefore semaphores are differentiable in terms of performance (mutexes are binary and some hardware has HW acceleration for this, and semaphores are more rich functionally but may have slower performance) as well as features.

3.4.3 MRAPI Reader/Writer locks

The MRAPI reader/writer locks provide a convenient mechanism for optimized access to critical sections that are not always intended to modify shared data. These primitives support multiple read-only accessors at any given time, or one exclusive accessor. This supports the RWL (Reader/Writer locks) software pattern that is commonly used for cases where there are more readers than writers. In order to guarantee fairness, MRAPI implementations must enforce serialization of requests such that that no new read lock will be granted while a blocking write lock request is pending.

3.5 MRAPI Memory

MRAPI supports two different notions of memory: *shared memory* and *remote memory*. Shared memory is provided in MRAPI to support applications that are deployed on hardware with physically shared memory with hardware managed cache coherency (*coherent shared memory*), but which cannot rely on a single operating system to provide a coherent shared memory allocation facility. It is

recognized that implementing this can be hard and currently discussions are ongoing with the MCA Hypervisor working group to understand a potential relationship for supporting coherent shared memory. Remote memory is provided for systems that require the use of explicit CPU, DMA, or other non-CPU mechanisms to move data between memory subsystems, or which do not support hardware managed cache coherency. The MRAPI specification allows for implementations to support only those types of MRAPI memory that are feasible for a given system, but the implementation must provide all API entry points and indicate via error reporting that a given request cannot be satisfied.

3.5.1 MRAPI Shared Memory

The functionality provided by the MRAPI shared memory API is similar to that of POSIX® shared memory, but MRAPI extends the functionality beyond the scope of a single operating system. It provides the ability to manage the access to physically coherent shared memory between heterogeneous threads of execution that may be on different operating systems and different types of cores.

3.5.2 MRAPI Remote Memory

Modern heterogeneous multicore systems often contain multiple memory spaces, where data is moved between memory spaces via non-CPU mechanisms such as direct memory access (DMA). One example is the Cell Broadband Engine processor: the Power Processor Element (PPE) is a standard PowerPC® core connected to main memory, but the processor also contains 8 synergistic processor elements (SPEs) which are each equipped with a small local store. Data must be copied to and from SPE local store via explicit DMA operations.

Remote memory might be implemented in many different ways, depending on the underlying hardware. Sometimes actual copying (i.e. read/write operations) are needed, sometimes just some software initiated cache operations are needed (invalidate/flush). However, the very purpose of an API should be to hide these differences in order to enable portable and hardware independent software. So in order to access data, some API call should be made that *might* boil down to either a "read", and "sync" or some combination (depending on the underlying hardware). The thing is that the user shouldn't really have to care. The software layer that constitutes the API should make sure that the necessary operations are performed, depending on the hardware.

From the point of view of a given processing element, remote memory is memory which cannot be accessed via standard load and store operations. For example, host memory is remote to a GPU core; the local store of a Cell SPE is remote to the other SPEs or the PPE.

MRAPI offers a set of API functions for manipulating remote memory. As with MRAPI shared memory, functions are provided for creating, initializing and attaching to remote memory. Unlike with MRAPI shared memory, the API provides functions for reading from and writing to remote memory.

The API does not place restrictions on the mechanism used for data transfer. However, catering to the common case where it is desirable to overlap data movement with computation, the API provides non-blocking read and write functions. In addition, flush and sync primitives are provided to allow support for software-managed caches. The API read and write functions also support scatter/gather style accesses.

For MRAPI users and implementers concerned about performance of the flush and synch functions, the MRAPI working group recommends use of multiple memory regions when the application writer is overly concerned; the implementation of the flush routine should have the semantics of "anything that is dirty should be pushed back to memory", versus “everything should be pushed back to memory”.

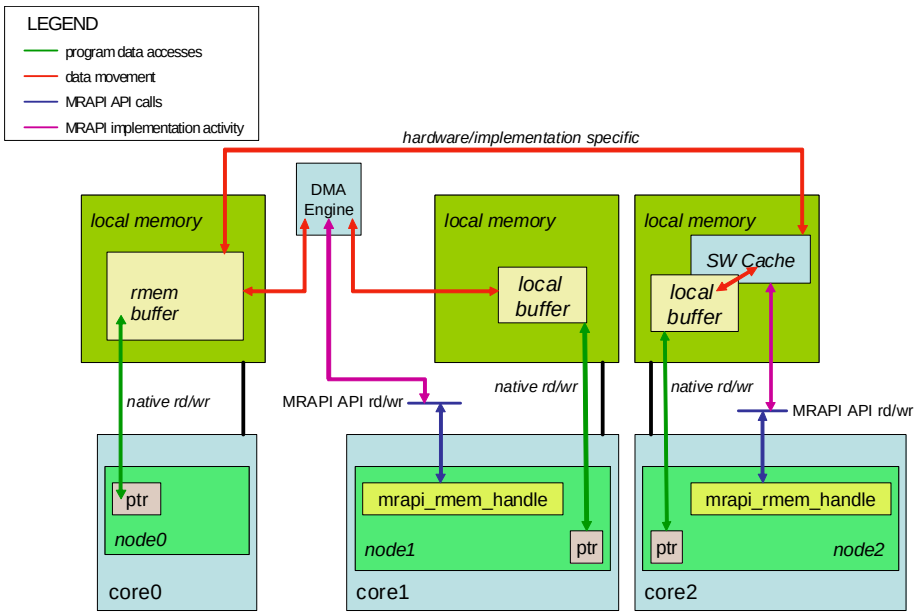

Figure 1 - Remote Memory Concepts

Figure 1 depicts the remote memory concepts in MRAPI. Access semantics are per rmem buffer instance, as follows:

- *Strict* semantics: the type of MRAPI access { DMA, sw cache, … } is defined at the time a rmem buffer is created. All MRAPI accesses to that rmem buffer must be of a uniform type. Each client of the buffer specifies an access type when on the get call and it is an error to request an access type other than that which was used to create the buffer.

- *Any* semantics: the type of MRAPI access {DMA, sw cache, …} is set to `MRAPI_RMEM_ATYPE_ANY` at the time the rmem buffer is created. When a client handle attaches it may specify any access type supported by the MRAPI implementation. Different types of accesses are supported concurrently. (Note that `MRAPI_RMEM_ATYPE_ANY` is only allowed for buffer creation, clients must call get using a specific access type, e.g., `MRAPI_RMEM_ATYPE_DEFAULT`, or other types provided by the implementation such as DMA, etc.)

Local pointer based read/write is always allowed (limited to access of local target buffers on clients). However, coherency issues must be managed by the application using MRAPI flush and synch calls (Sections 4.4.2.13 and 4.4.2.14). MRAPI implementations must guarantee that the effect of a synch operation must be complete before the next local read/write operation on the remote memory segment., and that the flush operation must block until it has completed.

Note that remote accesses (reads or writes) always results in a copy and must use MRAPI calls. Finally, implementations may define multiple access types (depending on underlying silicon capabilities), but must provide `MRAPI_RMEM_ATYPE_DEFAULT`, which has strict semantics and is guaranteed to work.

3.6 MRAPI Metadata

MRAPI provides a set of API calls designed to allow access to information regarding the underlying hardware context an application is running on. These capabilities are described in more detail in the following sections.

3.6.1 Metadata Resource Data Structure

A call to `mrapi_resources_get()` returns a data structure of type `mrapi_resource_t` , see Section 3.12.4. This data structure is provided in the form of a tree containing the set of resources that are visible to the calling MRAPI node. Each node in the tree represents a resource in the system, and each node contains attributes that provide additional information about a given resource. The resource tree may be optionally filtered by the `subsystem_filter` input parameter. Examples of such filters include CPU, cache, and hardware accelerators. An MRAPI implementation must define what filters it can provide as an enumerated type.

The resource data structure can contain hierarchical nodes in addition to the resource nodes themselves. For example, the concept of a core complex, which could contain multiple cores, would be represented as a parent node to the core nodes in the resource tree.

During initialization MRAPI may read in the system resources from a data file which may have a tree structure, such as XML or a device tree, so it is convenient to represent the resource data structure as a tree. Alternatively, the resources could be statically compiled in to the MRAPI implementation.

See section 6.1 for a use case and example code for getting and navigating a resource tree.

3.7 Attributes

Attributes are provided as a means of extension for the API. Different implementations may define and support additional attributes beyond those pre-defined by the API. In order to promote portability and implementation flexibility attributes are maintained in an opaque data structure, which may not be directly examined by the user. Each resource (e.g., mutex, semaphore, etc.) has an attributes data structure associated with it, and many attributes have a small set of predefined values which must be supported by MRAPI implementations The user may initialize, get and set these attributes.

If the user wants default behavior, then the intention is that they should not have to call the init/get/set attribute functions. However, if the user wants non-default behavior then the sequence of events should be as follows:

1) `mrapi_<`*`resource`*`>_init_attributes()`: returns an attributes structure with all attributes set to their default values.

2) `mrapi_<`*`resource`*`>_set_attribute():` (repeat for all attributes to be set): sets the given attribute in the attributes structure parameter to the given value.

3) `mrapi_<`*`resource`*`>_create():` you would pass in the attributes structure modified in the previous step as a parameter to this function.

Once a resource has been created, its attributes may not be changed.

At any time, the user can call `mrapi_<`*`resource`*`>_get_attribute()` to query the value of an attribute.

For a use case where attributes are customized, see section: 6.1.

3.8 Sharing Across Domains

Most of the MRAPI primitives are shared across MRAPI domains (Section 3.2) by default. We expect that implementations may potentially suffer a performance impact for resources that are shared across domains.

The following MRAPI primitives are shared across domains by default: mutexes, semaphores, reader/writer locks, and remote memory. For any of these primitives you can disable sharing across domains by setting the `MRAPI_DOMAIN_SHARED` attribute to `MRAPI_FALSE` and passing it to the corresponding `*_create()` function.

For the remaining primitive, e.g., MRAPI shared memory, the determination of which nodes it can be shared with (regardless of their domains) is specified in the nodes list that is passed in when the shared memory is created.

3.9 Waiting for Non-blocking Operations

The API has blocking, non-blocking, and single-attempt blocking variants for many functions. The non-blocking variants have "_i" appended to the function name to indicate that the function call will return *immediately* but the requested transaction will complete in a non-blocking manner. The single-attempt blocking functions will have the word "try" in the function name (for example, `mrapi_mutex_trylock`). Remote memory is the only resource that supports non-blocking variants (for reads/writes).

The non-blocking versions fill in an `mrapi_request_t` object and return control to the user before the requested operation is completed. The user can then use the `mrapi_test()`, `mrapi_wait()`, and `mrapi_wait_any()` functions to query the status of the non-blocking operation. The `mrapi_test()` function is non-blocking whereas the `mrapi_wait()` and `mrapi_wait_any()` functions will block until the requested operation completes or a timeout occurs.

Some blocking functions may have to wait for system events, e.g. buffer allocation or for data to arrive, and the duration of the blocking will be arbitrarily long (may be infinite), whereas other blocking functions don't have to wait for system events and can always complete in a timely fashion, with a success or failure. Single-attempt blocking functions that complete in this timely fashion include `mrapi_mutex_trylock()`, `mrapi_sem_trylock()`, `mrapi_rwl_trylock()`.

If a buffer of data is passed to a non-blocking operation (for example, to `mrapi_rmem_write_i()`) that buffer may not be accessed by the user application for the duration of the non-blocking operation. That is, once a buffer has been passed to a non-blocking operation, the program may not read or write the buffer until `mrapi_test()`, `mrapi_wait()`, or `mrapi_wait_any()` have indicated completion, or until `mrapi_cancel()` has canceled the operation.

3.10 MRAPI 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, some API functions allow implementations to *optionally* handle errors. Consistency 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 indicate success or failure which enables efficient coding practice. A parameter of type `mrapi_status_t` will encode success or failure states of API calls. `MRAPI_NULL` is a valid return value for `mrapi_status_t` (can be used for implementation optimization).

If a process/thread attached to a node were to fail it is generally up to the application to recover from this failure. MRAPI provides timeouts for the `mrapi_wait()` and `mrapi_wait_any()` functions and an `mrapi_cancel()` function to clear outstanding non-blocking requests (at the non-failing side). It is also possible to reinitialize a failed node, by first calling `mrapi_finalize()`.

3.11 MRAPI Timeout/Cancellation Philosophy

The MRAPI API provides timeout functionality for its non-blocking calls through the timeout capability of the `mrapi_wait()` and `mrapi_wait_any()` functions. All blocking functions implementations have `timeout_t` parameters. Setting the timeout to 0 means a function call will not time out. Setting it to MRAPI_INFINITE means it will eventually timeout but only after the maximum number of tries.

The MRAPI API also provides cancellation functionality for it's non-blocking calls through the `mrapi_cancel()` function.

3.12 MRAPI Datatypes

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

In general, API parameters that refer to MRAPI entities are *opaque handles* that should not be examined or interpreted by the application program. Obtaining a handle is done either via a *create* function or a *get* function. Create/get functions require MRAPI *ID* types (see Sections 3.12.1, 3.12.2, 3.12.4, 3.12.6, 3.12.13) to be passed in and will return a handle (see Sections 3.12.5, 3.12.7, 3.12.8, 3.12.10, 3.12.11) for use in all other function calls related to that MRAPI object.

3.12.1 mrapi_domain_t

The `mrapi_domain_t` type is used for MRAPI domains. The domain id scheme is implementation defined. For application portability we recommend using symbolic constants in your code. The `mrapi_domain_t` has an `mca_domain_t` equivalent.

3.12.2 mrapi_node_t

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

3.12.3 Initialization parameters and information

Initialization parameters were added to allow implementations to configure the MRAPI runtime. A parameter was also added to allow implementations to provide information about the MRAPI runtime both MRAPI specified and implementation specific information. For specifics on RCAPI information see below.

3.12.3.1 mrapi_param_t

Initialization parameters will vary by implementation, and may include specifications of the amount of resources to be used for a specific implementation or configuration, such as for example the maximum number of nodes, etc.

3.12.3.2 mrapi_info_t

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

MRAPI defined initialization information:

- `mrapi_version` -- MRAPI 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.
- `number_of_domains` -- Number of domains allowed by the implementation.
- `number_of_nodes` -- Number of nodes allowed by the implementation.

3.12.4 mrapi_resource_t

The `mrapi_resource_t` type is used to represent a resource in an MRAPI system. It is an opaque datatype with the exception of four elements: (1) `name`: a null-terminated C-style string containing the name of this resource, (2) `resource_type`: the type, (3) `children`: array of `mrapi_resource_t*`, and (4) `child_count` which indicates how many elements are in the children array. These two elements allow a set of resources to be arranged in a tree data structure which can be walked by the programmer using the `children` and `child_count` elements. The opaque section of the data structure contains attributes of the given resource. Access to attributes of the `mrapi_resource_t` type is through API calls defined in Section 4.6. A graphical representation of an `mrapi_resource_t` tree with a root node and two children is shown in Figure 2.

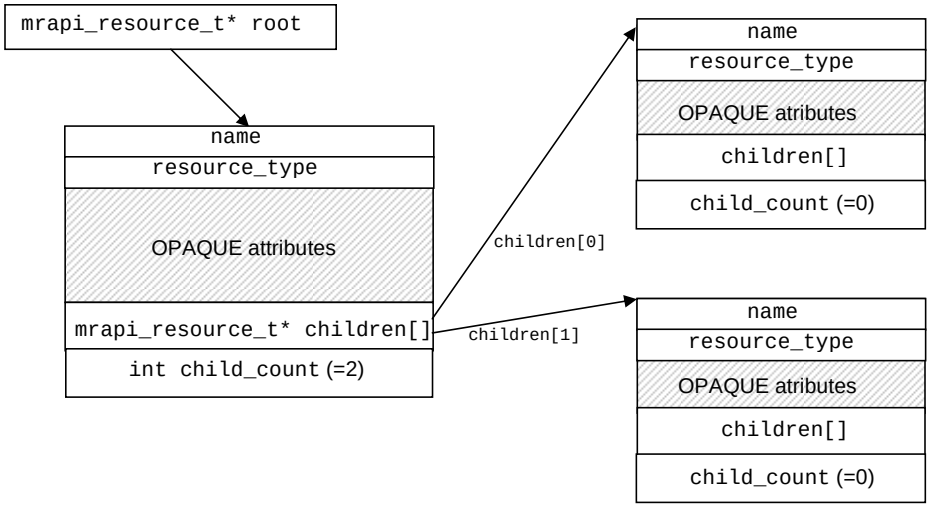

Figure 2 - An `mrapi_resource_t` Tree

3.12.5 mrapi_mutex_hndl_t

The `mrapi_mutex_hndl_t` type is used to lock and unlock a mutex. MRAPI routines for creating and using the `mrapi_mutex_hndl_t` type are covered in section 4.2.1. The `mrapi_mutex_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.6 mrapi_key_t

The `mrapi_key_t` type is used to support recursive locking and unlocking for mutexes (see section 4.3.1). The key is passed to the lock call and the system will fill in a unique key for that lock. The key is passed back on the unlock call.

3.12.7 mrapi_sem_hndl_t

The `mrapi_sem_hndl_t` type is used to lock and unlock a semaphore. MRAPI routines for creating and using the `mrapi_sem_hndl_t` type are covered in section 4.3.2. The `mrapi_sem_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.8 mrapi_rwl_hndl_t

The `mrapi_rwl_hndl_t` type is used to lock and unlock a reader/writer lock. MRAPI routines for creating and using the `mrapi_rwl_hndl_t` type are covered in section 4.3.3. The `mrapi_rwl_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.9 mrapi_rwl_mode_t

The `mrapi_rwl_mode_t` type is used to specify the type of reader/writer lock you are attempting to lock. The values are `MRAPI_READER` (shared) or `MRAPI_WRITER` (exclusive). See section 4.3.3 for the API calls that require this parameter.

3.12.10 mrapi_shmem_hndl_t

The `mrapi_shmem_hndl_t` type is used to access shared memory. MRAPI routines for creating and using the `mrapi_shmem_hndl_t` type are covered in section 4.4.1. The `mrapi_shmem_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.11 mrapi_rmem_hndl_t

The `mrapi_rmem_hndl_t` type is used to access remote memory. MRAPI routines for creating and using the `mrapi_rmem_hndl_t` type are covered in section 4.4.2. The `mrapi_rmem_hndl_t` is an opaque datatype whose exact definition is implementation defined.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.12 mrapi_rmem_atype_t

The `mrapi_rmem_atype_t` type is used to specify the access type to be used for remote memory (see Section 3.5.2 and Section 4.4.2). Access semantics are per remote memory buffer instance, and are either *strict* (meaning all clients must use the same access type), or *any* (meaning that clients may use any type supported by the MRAPI implementation). Implementations may define multiple access types (depending on underlying silicon capabilities), but must provide at minimum: `MRAPI_RMEM_ATYPE_ANY` (which indicates any semantics), and `MRAPI_RMEM_ATYPE_DEFAULT`, which has strict semantics. Note that `MRAPI_RMEM_ATYPE_ANY` is only valid for remote memory buffer creation, clients must use `MRAPI_RMEM_ATYPE_DEFAULT` or another specific type of access mechanism provided by the MRAPI implementation (for example DMA, etc.)

3.12.13 Identifiers: mrapi_mutex_id_t, mrapi_sem_id_t, mrapi_shmem_id_t, mrapi_rmem_id_t

The `mrapi_mutex_id_t, mrapi_sem_id_t, mrapi_shmem_id_t,` and `mrapi_rmem_id_t` types are used to get shared resources. The ID types are only used to get handles to the associated types of MRAPI entities.

- These ids may either be known *a priori* or passed as messages to the other nodes.
- The implementation defines what is "invalid". For any identifier, `mrapi_`*X*`_id` (for example `mrapi_mutex_id_t,` where *X*=`mutex`) there is a pair of corresponding identifiers in the MRAPI header file: `MRAPI_MAX_`*X*`_ID` and `MRAPI_MAX_USER_`*X*`_ID`, which can be examined by the application writer to determine valid ID ranges. MRAPI also supports `MRAPI_`*X*`_ID_ANY` (as in MCAPI endpoint creation). Thus, user specified ids can range from `0..MRAPI_MAX_USER_`*X*`_ID` and 'ANY' ids range from `MRAPI_MAX_USER_`*X*`_ID+1 .. MRAPI_MAX_`*X*`_ID`
- The user-specified space is disjoint from the ANY space to avoid race conditions for the user-specified ids.

3.12.14 Scalars: mrapi_uint64_t, mrapi_uint32_t, mrapi_uint16_t mrapi_uint8_t, mrapi_int64_t, mrapi_int32_t, mrapi_int16_t & mrapi_int8_t

The `mrapi_uint64_t`, `mrapi_uint32_t`, `mrapi_uint16_t`, `mrapi_uint8_t` , `mrapi_int64_t`, `mrapi_int32_t`, `mrapi_int16_t`, and `mrapi_int8_t` types are used for signed and unsigned 64-, 32-, 16-, and 8-bit scalars.

3.12.15 mrapi_request_t

The `mrapi_request_t` type is used to record the state of a pending non-blocking MRAPI transaction (see Section 4.5). Non-blocking MRAPI routines exist for only for reading and writing remote memory. An `mrapi_request_t` can only be used by the node it was created on. The `mrapi_request_t` has an `mca_request_t` equivalent.

NOTE: The MRAPI API user should not attempt to examine the contents of this datatype as this can result in non-portable application code.

3.12.16 mrapi_status_t

The `mrapi_status_t` type is an enumerated type used to record the result of an MRAPI API call. If a status can be returned by an API call, the associated MRAPI API call will allow a `mrapi_status_t` to be passed by reference. The API call will fill in the status code and the API user may examine the `mrapi_status_t` variable to determine the result of the call. The `mrapi_status_t` has an `mca_status_t` equivalent.

3.12.17 mrapi_timeout_t

The `mrapi_timeout_t` type is an unsigned scalar type used to indicate the duration that an `mrapi_wait()` or `mrapi_wait_any()` API call will block before reporting a timeout. The units of the `mrapi_timeout_t` datatype are implementation defined since mechanisms for time keeping vary from system to system. Applications should not rely on this feature for satisfaction of real time constraints as its usage will not guarantee application portability across MRAPI implementations. The `mrapi_timeout_t` datatype is intended only to allow for error detection and recovery. The `mrapi_timeout_t` has an `mca_timeout_t` equivalent. The reserved values are 0 for do not block at all, and MAX(unsigned 32-bit) for `MRAPI_INFINITE`.

3.12.18 Other MRAPI data types

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

3.13 MRAPI Compatibility with MCAPI

The MRAPI working group is following in the footsteps of the MCAPI working group. Therefore, this specification has adopted similar philosophies, and the same style for the API, datatypes, etc. Furthermore, since MRAPI and MCAPI are part of the larger Multicore Association Roadmap, the working group expended great effort to ensure that MRAPI functionality is orthogonal to MCAPI functionality while making sure they are interoperable (for example discussions around shared memory for MRAPI and zero copy messaging for MCAPI.)

3.14 Application Portability Concerns

The MRAPI working groups desires to enable application portability but cannot guarantee it. The guiding principles that should be used by application writers are (i) to write as much of the application in as portable a fashion as possible and (ii) encapsulating optimizations for efficiency or to take advantage of specialized dedicated hardware acceleration where possible and necessary. The end result of this approach should be that from a given MRAPI node's perspective it should not be possible nor required for that node to know whether it is interacting with another node within the same process, on the same processor, or even on the same chip. Furthermore, it should not be transparent to a given node whether it is interacting with another node that is implemented in hardware or software. The MRAPI working group believes that this approach will allow portability of software to be maintained at the interface level (e.g., the functional interface between nodes). However, the software implementation of a particular node cannot (and often should not) necessarily be preserved across a multicore SoC product line (or across product lines from different silicon providers) because a given node's functionality may be provided in different ways -- depending on the chosen multicore SoC. For more discussion on MRAPI nodes see Section 3.3.

3.15 MRAPI Implementation Concerns

This section provides guidance to implementers of MRAPI.

3.15.1 Thread Safe Implementations

MRAPI implementations are assumed to be reentrant (thread safe). Essentially, if an MRAPI implementation is available in a threaded environment, then it must be thread safe. MRAPI implementations can also be available in non-threaded environments. The provider of such implementations will need to clearly indicate that the implementation is not thread safe.

3.16 MRAPI Potential Future Extensions

With the goals of implementing MRAPI efficiently, the APIs are kept simple with potential for adding more functionality on top of MRAPI later. Some specific areas include read/copy/update (RCU) locks, non-owner remote memory allocation, application-level metadata, locking of resource lists, 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.

3.16.1 RCU (read/copy/update) locks

Although this feature is common in certain SMP operating systems, it is not clear that the feature scales well to embedded and/or non-SMP contexts. If research currently underway at various universities dispels this concern then RCU locks may be a feature worth adding to MRAPI.

3.16.2 Non-owner remote memory allocation of remote memory

Certain use cases considered by the working group indicated the usefulness of giving a node the ability to obtain memory from a different node. After consideration the working group determined that the API could be kept simple and this ability could be satisfied by using MCAPI messaging to allow one node to ask the other node to allocate on its behalf. In the future if this proves to be too inefficient for real-world application scenarios then this feature could be considered.

3.16.3 Application-level metadata

It is clear that application-level metadata can be used for rich higher-level functionality. The MRAPI working group believes this should be a layered service which can be built using a combination of MCAPI and MRAPI features. In case this proves to be difficult in the future we may wish to consider adding this feature.

3.16.4 Locking of resource lists

While similar APIs for resource management do provide functions for locking lists of resources, for now we believe this can be done well enough with mutexes and semaphores, especially given that MRAPI cannot enforce such locks (being a cooperative sharing API). If in the future it is proven we were mistaken this could be a feature we could consider adding.

3.16.5 Debug, Statistics and Status functions

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

3.16.6 Multiple Semaphore Lock Requests

It may be useful to add a feature that allows allocation of multiple counts of semaphore at once, instead of recursively calling the lock().

3.16.7 Node Lists for Remote Memory Creation Routines

We may wish to add a node list parameter to the shared memory creation routines. This would provide symmetry with the shared memory routines.

4 MRAPI API

The MRAPI API is divided into five major parts; (1) general API functions, (2) mutex, semaphore, and reader/writer lock functions, (3) memory-related functions, (4) metadata functions, and (5) non-blocking operations. The following sections enumerate the API calls for each of these five major parts.

4.1 Conventions

`MRAPI_IN` and `MRAPI_OUT` are used to distinguish between input and output parameters.

4.2 General

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

4.2.1 MRAPI_INITIALIZE

NAME

mrapi_initialize

SYNOPSIS

```
void mrapi_initialize(
  MRAPI_IN mrapi_domain_t domain_id,
  MRAPI_IN mrapi_node_t node_id,
  MRAPI_IN mrapi_parameters_t* mrapi_parameters,
  MRAPI_OUT mrapi_info_t* mrapi_info,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_initialize()` initializes the MRAPI environment on a given MRAPI node in a given MRAPI domain. It has to be called by each node using MRAPI. `mrapi_parameters` is used to pass implementation specific initialization parameters. `mrapi_info` is used to obtain information from the MRAPI implementation, including MRAPI and the underlying implementation version numbers, implementation vendor identification, the number of nodes in the topology, the number of ports on the local node and vendor specific implementation information, see the header files for additional 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 MRAPI node is an independent thread of control. An MRAPI node can call `mrapi_initialize()` once per node, and it is an error to call `mrapi_initialize()` multiple times from a given node, unless `mrapi_finalize()` is called in between. A given MRAPI 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.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ENO_INIT`	The MRAPI environment could not be initialized.
`MRAPI_ERR_NODE_INITIALIZED`	The MRAPI environment has already been initialized.
`MRAPI_ERR_NODE_INVALID`	The `node_id` parameter is not valid.
`MRAPI_ERR_DOMAIN_INVALID`	The `domain_id` parameter is not valid.
`MRAPI_ERR_PARAMETER`	Invalid mrapi_parameters or mrapi_info parameter.

NOTE

SEE ALSO

`mrapi_finalize()`

4.2.1.1 MRAPI_NODE_INIT_ATTRIBUTES

NAME

mrapi_node_init_attributes

SYNOPSIS

```
void mrapi_node_init_attributes(
  MRAPI_OUT mrapi_node_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Unless you want the defaults, this call must be used to initialize the values of an `mrapi_node_attributes_t` structure prior to `mrapi_node_set_attribute()`. Use `mrapi_node_set_attribute()` to change any default values prior to calling `mrapi_initialize()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attributes` parameter.

NOTE

SEE ALSO

4.2.1.2 MRAPI_NODE_SET_ATTRIBUTE

NAME

mrapi_node_set_attribute

SYNOPSIS

```
void mrapi_node_set_attribute(
  MRAPI_OUT mrapi_node_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_node_attributes_t` data structure prior to calling `mrapi_initialize()`. Calls to this function have no effect on node attributes once the node has been created/initialized.

At this time there are no MRAPI pre-defined node attributes.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size

NOTE

SEE ALSO

4.2.1.3 MRAPI_NODE_GET_ATTRIBUTE

NAME

mrapi_node_get_attribute

SYNOPSIS

```
void mrapi_node_get_attribute (
  MRAPI_IN mrapi_node_t node,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this node. The attribute may be viewed but may not be changed.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_node_set_attribute()` for a list of pre-defined attribute numbers.

4.2.2 MRAPI_FINALIZE

NAME

mrapi_finalize

SYNOPSIS

```
void mrapi_finalize(
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_finalize()` finalizes the MRAPI environment on a given MRAPI node and domain. It has to be called by each node using MRAPI. It is an error to call `mrapi_finalize()` without first calling `mrapi_initialize()`. An MRAPI node can call `mrapi_finalize()` once for each call to `mrapi_initialize()`, but it is an error to call `mrapi_finalize()` multiple times from a given <domain,node> unless `mrapi_initialize()` has been called prior to each `mrapi_finalize()` call.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_NODE_FINALFAILED`	The MRAPI environment could not be finalized.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.2.3 MRAPI_DOMAIN_ID_GET

NAME

mrapi_domain_id_get

SYNOPSIS

```
mrapi_domain_t mrapi_domain_id_get(
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the domain id associated with the local node.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.2.4 MRAPI_NODE_ID_GET

NAME

mrapi_node_id_get

SYNOPSIS

```
mrapi_node_t mrapi_node_id_get(
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the node id associated with the local node and domain.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3 MRAPI Synchronization Primitives

MRAPI supports three types of synchronization primitives: mutexes, semaphores and reader/writer locks. They provide locking functionality through the use of a flag (mutex) or a counter (semaphores) or combination of flag and counter (reader/writer locks). Although a binary semaphore can be used as a mutex, MRAPI explicitly provides mutexes to allow for hardware acceleration. Although Reader/Writer locks can be implemented on top of mutexes and semaphores, MRAPI provides them as a convenience.

Within MRAPI, there is no concept of ownership for the synchronization primitives. Any node may create or get a mutex, semaphore or reader/writer lock (provided it knows the shared key) and any node may delete the mutex, semaphore or reader/writer lock. To support performance/debuggability tradeoffs, MRAPI provides two types of error checking; basic (default) and extended (enabled via the `MRAPI_ERROR_EXT` attribute). When extended error checking is enabled, if lock is called on a mutex, semaphore or reader/writer lock that no longer exists, an MRAPI_ERR_[MUTEX|SEM|RWL]_DELETED error code will be returned. When extended error checking is disabled, the MRAPI_ERR_[MUTEX | SEM | RWL]_INVALID error will be returned and the lock will fail. The benefit of extended error checking is for early functional verification/validation of the code and the working group feels this is a valuable feature for easing the burden of multicore development and debugging. Because extended error checking can be resource intensive, it is optional and disabled by default.

By default, the synchronization primitives are shared across domains. Set the MRAPI_DOMAIN_SHARED attribute to false when you create the mutex, semaphore or reader/writer lock to disable resource sharing across domains. Note that we can not always expect sharing across domains to be efficient.

4.3.1 Mutexes

MRAPI mutexes provide exclusive locking functionality through the use of a flag (just like a binary semaphore). MRAPI mutexes support recursive locking. Recursive locking means that once a mutex is locked, lock may be called again before unlock is called. For each call to lock, a unique lock key is returned. This lock key must be passed in to the call to unlock. The implementation uses the keys to match the order of the lock/unlock calls. Recursive locking is disabled by default and can be enabled by setting the `MRAPI_MUTEX_RECURSIVE` attribute when the mutex is created. When the mutex is not recursive, the lock_keys are ignored.

If `mrapi_mutex_lock()` is called and the lock is currently locked and recursive locking is disabled, then the function will block until the lock is available. It is safer to use `mrapi_mutex_trylock()` unless you are certain that the lock will eventually succeed. Otherwise, a thread of execution can block forever waiting for the lock.

4.3.1.1 MRAPI_MUTEX_CREATE

NAME

mrapi_mutex_create

SYNOPSIS

```
mrapi_mutex_hndl_t mrapi_mutex_create(
  MRAPI_IN mrapi_mutex_id_t mutex_id,
  MRAPI_IN mrapi_mutex_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function creates a mutex. For non-default behavior, attributes must be set before the call to `mrapi_mutex_create()`. Once a mutex has been created, its attributes may not be changed. If the attributes are `NULL`, then default attributes will be used. The recursive attribute is disabled by default. If you want to enable recursive locking/unlocking then you need to set that attribute before the call to create. If `mutex_id` is set to `MRAPI_MUTEX_ID_ANY`, then MRAPI will choose an internal id for you.

RETURN VALUE

On success a mutex handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the mutex already exists, status will be set to `MRAPI_EXISTS` and the handle returned will not be a valid handle.

ERRORS

`MRAPI_ERR_MUTEX_ID_INVALID`	The `mutex_id` is not a valid mutex id.
`MRAPI_ERR_MUTEX_EXISTS`	This mutex is already created.
`MRAPI_ERR_MUTEX_LIMIT`	Exceeded maximum number of mutexes allowed.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_PARAMETER`	Invalid attributes parameter.

NOTE

SEE ALSO

See `mrapi_mutex_init_attributes()` and `mrapi_mutex_set_attribute()`
See datatypes identifiers discussion: 3.12.13

4.3.1.2 MRAPI_MUTEX_INIT_ATTRIBUTES

NAME

mrapi_mutex_init_attributes

SYNOPSIS

```
void mrapi_mutex_init_attributes(
  MRAPI_OUT mrapi_mutex_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function initializes the values of an `mrapi_mutex_attributes_t` structure. For non-default behavior this function should be called prior to calling `mrapi_mutex_set_attribute()`. You would then use `mrapi_mutex_set_attribute()` to change any default values prior to calling `mrapi_mutex_create()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.1.3 MRAPI_MUTEX_SET_ATTRIBUTE

NAME

mrapi_mutex_set_attribute

SYNOPSIS

```
void mrapi_mutex_set_attribute (
  MRAPI_OUT mrapi_mutex_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_mutex_attributes_t` data structure prior to calling `mrapi_mutex_create()`. Calls to this function have no effect on mutex attributes once the mutex has been created.

MRAPI pre-defined mutex attributes:

`Attribute num:`	Description:	Datatype:	Default:
`MRAPI_MUTEX_RECURSIVE`	Indicates whether or not this is a recursive mutex.	`mrapi_boolean_t`	`MRAPI_FALSE`
`MRAPI_ERROR_EXT`	Indicates whether or not this mutex has extended error checking enabled.	`mrapi_boolean_t`	`MRAPI_FALSE`
`MRAPI_DOMAIN_SHARED`	Indicates whether or not the mutex is shareable across domains.	`mrapi_boolean_t`	`MRAPI_TRUE`

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_ATTR_READONLY`	Attribute can not be modified.
`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.1.4 MRAPI_MUTEX_GET_ATTRIBUTE

NAME

mrapi_mutex_get_attribute

SYNOPSIS

```
void mrapi_mutex_get_attribute (
  MRAPI_IN mrapi_mutex_hndl_t mutex,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this mutex. The attributes may be viewed but may not be changed (for this mutex).

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter. When extended error checking is enabled, if this function is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_MUTEX_INVALID`	Argument is not a valid mutex handle.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_mutex_set_attribute()` for a list of pre-defined attribute numbers.

4.3.1.5 MRAPI_MUTEX_GET

NAME

mrapi_mutex_get

SYNOPSIS

```
mrapi_mutex_hndl_t mrapi_mutex_get(
  MRAPI_IN mrapi_mutex_id_t mutex_id,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Given a `mutex_id`, this function returns the MRAPI handle for referencing that mutex.

RETURN VALUE

On success the mutex handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_MUTEX_ID_INVALID`	The `mutex_id` parameter does not refer to a valid mutex or it is set to MRAPI_MUTEX_ID_ANY.
`MRAPI_ERR_NODE_NOTINIT`	The node/domain is not initialized.
`MRAPI_ERR_DOMAIN_NOTSHARED`	This resource can not be shared by this domain.
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_ID_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

See `mrapi_mutex_set_attribute()`.

4.3.1.6 MRAPI_MUTEX_DELETE

NAME

mrapi_mutex_delete

SYNOPSIS

```
void mrapi_mutex_delete(
  MRAPI_IN mrapi_mutex_hndl_t mutex,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function deletes the mutex. The mutex may only be deleted if it is unlocked. If the mutex attributes indicate extended error checking is enabled then all subsequent lock requests will be notified that the mutex was deleted. When extended error checking is enabled, if this function is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_MUTEX_INVALID`	Argument is not a valid mutex handle.
`MRAPI_ERR_MUTEX_LOCKED`	The mutex is locked and cannot be deleted.
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.1.7 MRAPI_MUTEX_LOCK

NAME

mrapi_mutex_lock

SYNOPSIS

```
void mrapi_mutex_lock (
  MRAPI_IN mrapi_mutex_hndl_t mutex,
  MRAPI_OUT mrapi_key_t* lock_key,
  MRAPI_IN mrapi_timeout_t timeout,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to lock a mutex and will block if another node has a lock on the mutex. When it obtains the lock, it sets up a unique key for that lock and that key is to be passed back on the call to unlock. This key allows us to support recursive locking. The `lock_key` is only valid if status indicates success. Whether or not a mutex can be locked recursively is controlled via the `MRAPI_MUTEX_RECURSIVE` attribute, and the default is `MRAPI_FALSE`.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_MUTEX_INVALID`	Argument is not a valid mutex handle.
`MRAPI_ERR_MUTEX_LOCKED`	Mutex is already locked by another node or mutex is already locked by this node and is not a recursive mutex.
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_INVALID`.
`MRAPI_TIMEOUT`	Timeout was reached.
`MRAPI_ERR_PARAMETER`	Invalid lock_key or timeout parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.1.8 MRAPI_MUTEX_TRYLOCK

NAME

mrapi_mutex_trylock

SYNOPSIS

```
mrapi_boolean_t mrapi_mutex_trylock(
  MRAPI_IN mrapi_mutex_hndl_t mutex,
  MRAPI_OUT mrapi_key_t* lock_key,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to obtain a lock on the mutex. If the lock can't be obtained because it is already locked by another node then the function will immediately return `MRAPI_FALSE` and `status will be set to MRAPI_SUCCESS`. If the request can't be satisfied for any other reason, then this function will immediately return MRAPI_FALSE and status will be set to the appropriate error code below. If it is successful in obtaining the lock, it sets up a unique key for that lock and that key is to be passed back on the call to unlock. The `lock_key` is only valid if status indicates success and the function returns `MRAPI_TRUE`. This key allows us to support recursive locking. Whether or not a mutex can be locked recursively is controlled via the `MRAPI_MUTEX_RECURSIVE` attribute, and the default is `MRAPI_FALSE`.

RETURN VALUE

Returns `MRAPI_TRUE` if the lock was acquired, returns `MRAPI_FALSE` otherwise. If there was an error then `*status` will be set to indicate the error from the table below, otherwise `*status` will indicate `MRAPI_SUCCESS`. If the lock could not be obtained then `*status` will be either `MRAPI_ELOCKED` or one of the error conditions in the table below. When extended error checking is enabled, if lock is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned and the lock will fail.

ERRORS

`MRAPI_ERR_MUTEX_INVALID`	Argument is not a valid mutex handle.
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_INVALID`.
`MRAPI_ERR_MUTEX_LOCKED`	Mutex is already locked by another node or mutex is already locked by this node and is not a recursive mutex.
`MRAPI_ERR_PARAMETER`	Invalid lock_key parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.1.9 MRAPI_MUTEX_UNLOCK

NAME

mrapi_mutex_unlock

SYNOPSIS

```
void mrapi_mutex_unlock(
  MRAPI_IN mrapi_mutex_hndl_t mutex,
  MRAPI_IN mrapi_key_t* lock_key,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function unlocks a mutex. If the mutex is recursive, then the `lock_key` parameter passed in must match the `lock_key` that was returned by the corresponding call to lock the mutex, and the set of recursive locks must be released using `lock_keys` in the reverse order that they were obtained. When extended error checking is enabled, if this function is called on a mutex that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_MUTEX_INVALID` error will be returned.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_MUTEX_INVALID`	Argument is not a valid mutex handle.
`MRAPI_ERR_MUTEX_NOTLOCKED`	Mutex is not locked.
`MRAPI_ERR_MUTEX_KEY`	`lock_key` is invalid for this mutex.
`MRAPI_ERR_MUTEX_LOCKORDER`	The unlock call does not match the lock order for this recursive mutex.
`MRAPI_ERR_PARAMETER`	Invalid `lock_key` parameter.
`MRAPI_ERR_MUTEX_DELETED`	If the mutex has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_MUTEX_DELETED` otherwise MRAPI will just return `MRAPI_ERR_MUTEX_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2 Semaphores

MRAPI semaphores provide shared locking functionality through the use of a counter. When an MRAPI semaphore is created, the maximum number of available locks is specified (in the `shared_lock_limit` parameter). If `mrapi_sem_lock()` is called and all locks are currently locked, then the function will block until a lock is available. It is safer to use `mrapi_sem_trylock()` unless you are certain that the lock will eventually succeed. Otherwise, your thread of execution can block forever waiting for the lock.

4.3.2.1 MRAPI_SEM_CREATE

NAME

mrapi_sem_create

SYNOPSIS

```
mrapi_sem_hndl_t mrapi_sem_create(
  MRAPI_IN mrapi_sem_id_t sem_id,
  MRAPI_IN mrapi_sem_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t shared_lock_limit,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function creates a semaphore. Unless you want the defaults, attributes must be set before the call to `mrapi_sem_create()`. Once a semaphore has been created, its attributes may not be changed. If the attributes are NULL, then implementation defined default attributes will be used. If `sem_id` is set to `MRAPI_SEM_ID_ANY`, then MRAPI will choose an internal id for you. The `shared_lock_limit` parameter indicates the maximum number of available locks and it must be between 0 and MRAPI_MAX_SEM_SHAREDLOCKS.

RETURN VALUE

On success a semaphore handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the semaphore already exists, status will be set to `MRAPI_EXISTS` and the handle returned will not be a valid handle.

ERRORS

`MRAPI_ERR_SEM_ID_INVALID`	The `semaphore_id` is not a valid semaphore id.
`MRAPI_ERR_SEM_EXISTS`	This semaphore is already created.
`MRAPI_ERR_SEM_LIMIT`	Exceeded maximum number of semaphores allowed.
`MRAPI_ERR_SEM_LOCKLIMIT`	The shared lock limit is out of bounds.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_PARAMETER`	Invalid attributes parameter.

NOTE

SEE ALSO

See `mrapi_sem_init_attributes()` and `mrapi_sem_set_attribute()`.

See also datatypes identifiers discussion: 3.12.13

4.3.2.2 MRAPI_SEM_INIT_ATTRIBUTES

NAME

mrapi_sem_init_attributes

SYNOPSIS

```
void mrapi_sem_init_attributes(
  MRAPI_OUT mrapi_sem_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Unless you want the defaults, this function should be called to initialize the values of an `mrapi_sem_attributes_t` structure prior to `mrapi_sem_set_attribute()`. You would then use `mrapi_sem_set_attribute()` to change any default values prior to calling `mrapi_sem_create()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attributes parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2.3 MRAPI_SEM_SET_ATTRIBUTE

NAME

mrapi_sem_set_attribute

SYNOPSIS

```
void mrapi_sem_set_attribute(
  MRAPI_OUT mrapi_sem_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_sem_attributes_t` data structure prior to calling `mrapi_sem_create()`. Calls to this function have no effect on semaphore attributes once the semaphore has been created.

MRAPI pre-defined semaphore attributes:

`Attribute num:`	Description:	Datatype:	Default:
`MRAPI_ERROR_EXT`	Indicates whether or not this semaphore has extended error checking enabled.	`mrapi_boolean_t`	`MRAPI_FALSE`
`MRAPI_DOMAIN_SHARED`	Indicates whether or not this semaphore is shareable across domains.	`mrapi_boolean_t`	`MRAPI_TRUE`

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_ATTR_READONLY`	Attribute can not be modified.
`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2.4 MRAPI_SEM_GET_ATTRIBUTE

NAME

mrapi_sem_get_attribute

SYNOPSIS

```
void mrapi_sem_get_attribute (
  MRAPI_IN mrapi_sem_hndl_t sem,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this semaphore. The attribute may be viewed but may not be changed (for this semaphore).

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter. When extended error checking is enabled, if this function is called on a semaphore that no longer exists, an `MRAPI_ERR_MUTEX_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_SEM_INVALID`	Argument is not a valid semaphore handle.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_sem_set_attribute()` for a list of pre-defined attribute numbers.

4.3.2.5 MRAPI_SEM_GET

NAME

mrapi_sem_get

SYNOPSIS

```
mrapi_sem_hndl_t mrapi_sem_get(
  MRAPI_IN mrapi_sem_id_t sem_id,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Given a `sem_id`, this function returns the MRAPI handle for referencing that semaphore.

RETURN VALUE

On success the semaphore handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a semaphore that no longer exists, an `MRAPI_ERR_SEM_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_SEM_ID_INVALID`	The `sem_id` parameter does not refer to a valid semaphore or was called with `sem_id` set to MRAPI_SEM_ID_ANY.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_DOMAIN_NOTSHARED`	This resource can not be shared by this domain.
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_ID_INVALID`.

NOTE

SEE ALSO

See `mrapi_sem_set_attribute()`

4.3.2.6 MRAPI_SEM_DELETE

NAME

mrapi_sem_delete

SYNOPSIS

```
void mrapi_sem_delete(
  MRAPI_IN mrapi_sem_hndl_t sem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function deletes the semaphore. The semaphore will only be deleted if the semaphore is not locked. If the semaphore attributes indicate extended error checking is enabled then all subsequent lock requests will be notified that the semaphore was deleted.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a semaphore that no longer exists, an `MRAPI_ERR_SEM_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_SEM_INVALID`	Argument is not a valid semaphore handle.
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_INVALID`.
`MRAPI_ERR_SEM_LOCKED`	The semaphore is locked and cannot be deleted.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2.7 MRAPI_SEM_LOCK

NAME

mrapi_sem_lock

SYNOPSIS

```
void mrapi_sem_lock(
  MRAPI_IN mrapi_sem_hndl_t sem,
  MRAPI_IN mrapi_timeout_t timeout,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to obtain a single lock on the semaphore and will block until a lock is available or the timeout is reached (if timeout is non-zero). If the request can't be satisfied for some other reason, this function will return the appropriate error code below. An application may make this call as many times as needed to obtain multiple locks, up to the limit specified by the `shared_lock_limit` parameter used when the semaphore was created.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if lock is called on semaphore that no longer exists, an `MRAPI_ERR_SEM_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned and the lock will fail.

ERRORS

`MRAPI_ERR_SEM_INVALID`	Argument is not a valid semaphore handle.
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_INVALID`.
`MRAPI_TIMEOUT`	Timeout was reached.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2.8 MRAPI_SEM_TRYLOCK

NAME

mrapi_sem_trylock

SYNOPSIS

```
mrapi_boolean_t mrapi_sem_trylock(
   MRAPI_IN mrapi_sem_hndl_t sem,
   MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to obtain a single lock on the semaphore. If the lock can't be obtained because all the available locks are already locked (by this node and/or others) then the function will immediately return `MRAPI_FALSE and status will be set to MRAPI_SUCCESS`. If the request can't be satisfied for any other reason, then this function will immediately return MRAPI_FALSE and status will be set to the appropriate error code below.

RETURN VALUE

Returns `MRAPI_TRUE` if the lock was acquired, returns `MRAPI_FALSE` otherwise. If there was an error then `*status` will be set to indicate the error from the table below, otherwise `*status` will indicate `MRAPI_SUCCESS`. If the lock could not be obtained then `*status` will be either `MRAPI_ELOCKED` or one of the error conditions in the table below. When extended error checking is enabled, if this function is called on a semaphore that no longer exists, an `MRAPI_ERR_SEM_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_SEM_INVALID`	Argument is not a valid semaphore handle.
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.2.9 MRAPI_SEM_UNLOCK

NAME

mrapi_sem_unlock

SYNOPSIS

```
void mrapi_sem_unlock (
  MRAPI_IN mrapi_sem_hndl_t sem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function releases a single lock.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a semaphore that no longer exists, an `MRAPI_ERR_SEM_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_SEM_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_SEM_INVALID`	Argument is not a valid semaphore handle.
`MRAPI_ERR_SEM_NOTLOCKED`	This node does not have a lock on this semaphore
`MRAPI_ERR_SEM_DELETED`	If the semaphore has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_SEM_DELETED` otherwise MRAPI will just return `MRAPI_ERR_SEM_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3 Reader/Writer Locks

MRAPI reader/writer locks provide a combination of exclusive (writer) and shared (reader) locking functionality. A single reader/writer lock provides both types of locking. The type of lock desired is passed in the mode parameter to the lock function.

4.3.3.1 MRAPI_RWL_CREATE

NAME

mrapi_rwl_create

SYNOPSIS

```
mrapi_rwl_hndl_t mrapi_rwl_create(
  MRAPI_IN mrapi_rwl_id_t rwl_id,
  MRAPI_IN mrapi_rwl_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t reader_lock_limit,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function creates a reader/writer lock. Unless you want the defaults, attributes must be set before the call to `mrapi_rwl_create()`. Once a reader/writer lock has been created, its attributes may not be changed. If the attributes are NULL, then implementation defined default attributes will be used. If `rwl_id` is set to `MRAPI_RWL_ID_ANY`, then MRAPI will choose an internal id for you.

RETURN VALUE

On success a reader/writer lock handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the reader/writer lock already exists, status will be set to `MRAPI_EXISTS` and the handle returned will not be a valid handle.

ERRORS

`MRAPI_ERR_RWL_ID_INVALID`	The `rwl_id` is not a valid reader/writer lock id.
`MRAPI_ERR_RWL_EXISTS`	This reader/writer lock is already created.
`MRAPI_ERR_RWL_LIMIT`	Exceeded maximum number of reader/writer locks allowed.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_PARAMETER`	Invalid `attributes` parameter.

NOTE

SEE ALSO

See `mrapi_rwl_init_attributes()` and `mrapi_rwl_set_attribute()`.
See datatypes identifiers discussion: Section 3.12.13

4.3.3.2 MRAPI_RWL_INIT_ATTRIBUTES

NAME

mrapi_rwl_init_attributes

SYNOPSIS

```
void mrapi_rwl_init_attributes(
  MRAPI_OUT mrapi_rwl_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Unless you want the defaults, this call must be used to initialize the values of an `mrapi_rwl_attributes_t` structure prior to `mrapi_rwl_set_attribute()`. Use `mrapi_rwl_set_attribute()` to change any default values prior to calling `mrapi_rwl_create()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3.3 MRAPI_RWL_SET_ATTRIBUTE

NAME

mrapi_rwl_set_attribute

SYNOPSIS

```
void mrapi_rwl_set_attribute(
  MRAPI_OUT mrapi_rwl_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_rwl_attributes_t` data structure prior to calling `mrapi_rwl_create()`. Calls to this function have no effect on mutex attributes once the mutex has been created.

MRAPI pre-defined reader/writer lock attributes:

`Attribute num:`	Description:	Datatype:	Default:
`MRAPI_ERROR_EXT`	Indicates whether or not this reader/writer lock has extended error checking enabled.	`mrapi_boolean_t`	`MRAPI_FALSE`
`MRAPI_DOMAIN_SHARED`	Indicates whether or not the reader/writer lock is shareable across domains.	`mrapi_boolean_t`	`MRAPI_TRUE`

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_ATTR_READONLY`	Attribute can not be modified.
`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3.4 MRAPI_RWL_GET_ATTRIBUTE

NAME

mrapi_rwl_get_attribute

SYNOPSIS

```
void mrapi_rwl_get_attribute (
  MRAPI_IN mrapi_rwl_hndl_t rwl,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this reader/writer lock. The attribute may be viewed but may not be changed (for this reader/writer lock).

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter. When extended error checking is enabled, if this function is called on a reader/writer lock that no longer exists, an `MRAPI_ERR_RWL_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_RWL_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid attribute parameter.
`MRAPI_ERR_RWL_INVALID`	Argument is not a valid rwl handle.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

It is up to the implementation as to whether a reader/writer lock may be shared across domains. This is specified as an attribute during creation and the default is `MRAPI_FALSE`.

SEE ALSO

`mrapi_rwl_set_attribute()` for a list of pre-defined attribute numbers.

4.3.3.5 MRAPI_RWL_GET

NAME

mrapi_rwl_get

SYNOPSIS

```
mrapi_rwl_hndl_t mrapi_rwl_get(
  MRAPI_IN mrapi_rwl_id_t rwl_id,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Given a `rwl_id`, this function returns the MRAPI handle for referencing that reader/writer lock.

RETURN VALUE

On success the reader/writer lock handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RWL_ID_INVALID`	The `rwl_id` parameter does not refer to a valid reader/writer lock or it was called with `rwl_id` set to MRAPI_RWL_ID_ANY.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_DOMAIN_NOTSHARED`	This resource can not be shared by this domain.
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_ID_INVALID`.

NOTE

SEE ALSO

See `mrapi_rwl_set_attribute()`

4.3.3.6 MRAPI_RWL_DELETE

NAME

mrapi_rwl_delete

SYNOPSIS

```
void mrapi_rwl_delete(
  MRAPI_IN mrapi_rwl_hndl_t rwl,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function deletes the reader/writer lock. A reader/writer lock can only be deleted if it is not locked. If the reader/writer lock attributes indicate extended error checking is enabled then all subsequent lock requests will be notified that the reader/writer lock was deleted.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a reader/writer lock that no longer exists, an `MRAPI_ERR_RWL_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_RWL_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_RWL_INVALID`	Argument is not a valid reader/writer lock handle.
`MRAPI_ERR_RWL_LOCKED`	The reader/writer lock was locked and cannot be deleted.
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3.7 MRAPI_RWL_LOCK

NAME

mrapi_rwl_lock

SYNOPSIS

```
void mrapi_rwl_lock(
  MRAPI_IN mrapi_rwl_hndl_t rwl,
  MRAPI_IN mrapi_rwl_mode_t mode,
  MRAPI_IN mrapi_timeout_t timeout,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to obtain a single lock on the reader/writer lock and will block until a lock is available or the timeout is reached (if timeout is non-zero). A node may only have one reader lock or one writer lock at any given time. The `mode` parameter is used to specify the type of lock: `MRAPI_READER` (shared) or `MRAPI_WRITER` (exclusive). If the lock can't be obtained for some other reason, this function will return the appropriate error code below.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if lock is called on a reader/writer lock that no longer exists, an `MRAPI_ERR_RWL_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_RWL_INVALID` error will be returned. In both cases the attempt to lock will fail.

ERRORS

`MRAPI_ERR_RWL_INVALID`	Argument is not a valid reader/writer lock handle.
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_INVALID`.
`MRAPI_TIMEOUT`	Timeout was reached.
`MRAPI_ERR_RWL_LOCKED`	The caller already has a lock
`MRAPI_ERR_PARAMETER`	Invalid mode.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3.8 MRAPI_RWL_TRYLOCK

NAME

mrapi_rwl_trylock

SYNOPSIS

```
mrapi_boolean_t mrapi_rwl_trylock(
  MRAPI_IN mrapi_rwl_hndl_t rwl,
  MRAPI_IN mrapi_rwl_mode_t mode,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attempts to obtain a single lock on the reader/writer lock. A node may only have one reader lock or one writer lock at any given time. The mode parameter is used to specify the type of lock: `MRAPI_READER` (shared) or `MRAPI_WRITER` (exclusive). If the lock can't be obtained because a reader lock was requested and there is already a writer lock or a writer lock was requested and there is already any lock then the function will immediately return `MRAPI_FALSE` and status will be set to `MRAPI_SUCCESS`. If the request can't be satisfied for any other reason, then this function will immediately return MRAPI_FALSE and status will be set to the appropriate error code below.

RETURN VALUE

Returns `MRAPI_TRUE` if the lock was acquired, returns `MRAPI_FALSE` otherwise. If there was an error then `*status` will be set to indicate the error from the table below, otherwise `*status` will indicate `MRAPI_SUCCESS`. If the lock could not be obtained then `*status` will be either `MRAPI_ELOCKED` or one of the error conditions in the table below. When extended error checking is enabled, if `trylock` is called on a reader/writer lock that no longer exists, an `MRAPI_ERR_RWL_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_RWL_INVALID` error will be returned and the lock will fail.

ERRORS

`MRAPI_ERR_RWL_INVALID`	Argument is not a valid reader/writer lock handle.
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_INVALID`.
`MRAPI_ERR_RWL_LOCKED`	The reader/writer lock is already exclusively locked.
`MRAPI_ERR_PARAMETER`	Invalid mode.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.3.3.9 MRAPI_RWL_UNLOCK

NAME

mrapi_rwl_unlock

SYNOPSIS

```
void mrapi_rwl_unlock (
  MRAPI_IN mrapi_rwl_hndl_t rwl,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function releases a single lock. The lock to be released will be either a reader lock or a writer lock, as specified by the mode parameter used when the lock was obtained.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. When extended error checking is enabled, if this function is called on a reader/writer lock that no longer exists, an `MRAPI_ERR_RWL_DELETED` error code will be returned. When extended error checking is disabled, the `MRAPI_ERR_RWL_INVALID` error will be returned.

ERRORS

`MRAPI_ERR_RWL_INVALID`	Argument is not a valid reader/writer lock handle.
`MRAPI_ERR_RWL_NOTLOCKED`	This node does not currently hold the given type (reader/writer) of lock.
`MRAPI_ERR_RWL_DELETED`	If the reader/writer lock has been deleted then if `MRAPI_ERROR_EXT` attribute is set, MRAPI will return `MRAPI_ERR_RWL_DELETED` otherwise MRAPI will just return `MRAPI_ERR_RWL_INVALID`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4 MRAPI Memory

MRAPI supports two memory concepts: shared memory and remote memory. Shared memory is semantically the same as shared memory in, e.g., POSIX® except that it is also supported for heterogeneous systems (here heterogeneity may mean hardware or software), otherwise there would be no need to have it in the MRAPI standard. Remote memory caters to non-uniform memory architecture machines such as the Cell processor, where the SPEs cannot access PPE main memory via load and store instructions, and must use DMA or a software cache, or special purpose accelerators such as graphics processing units which also use DMA.

For both memory types, remote and shared, a node must attach before using the memory and detach when finished.

4.4.1 Shared Memory

MRAPI shared memory provides functionality to create and get shared memory segments, attach them to the application's private memory space, query the memory attributes and detach and delete the memory segments. For a detailed description of MRAPI memory semantics refer to Section 3.5. The minimum MRAPI shared memory is considered application/user-level; implementations could defined additional attributes which specify various privilege levels but this should be used with caution as it can seriously inhibit application portability.

For shared memory, MRAPI allows the creator of the memory handle to specify which nodes are allowed to access the shared memory region. In some cases this will cause MRAPI to return an error code if the request cannot be satisfied. An example of this would be the IBM Cell processor in which the main core and the dedicated processing engines do not have access to physically shared memory.

4.4.1.1 MRAPI SHMEM_CREATE

NAME

mrapi_shmem_create

SYNOPSIS

```
mrapi_shmem_hndl_t mrapi_shmem_create(
   MRAPI_IN mrapi_shmem_id_t shmem_id,
   MRAPI_IN mrapi_uint_t size,
   MRAPI_IN mrapi_node_t* nodes,
   MRAPI_IN mrapi_uint_t nodes_size,
   MRAPI_IN mrapi_shmem_attributes_t* attributes,
   MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function creates a shared memory segment. The `size` parameter specifies the size of the shared memory region in bytes. Unless you want the defaults, attributes must be set before the call to `mrapi_shmem_create()`. A list of nodes that can access the shared memory can be passed in the nodes parameter and `nodes_size` should contain the number of nodes in the list. If nodes is NULL, then all nodes will be allowed to access the shared memory. Once a shared memory segment has been created, its attributes may not be changed. If the attributes parameter is NULL, then implementation defined default attributes will be used. In the case where the shared memory segment already exists, status will be set to `MRAPI_EXISTS` and the handle returned will not be a valid handle. If `shmem_id` is set to `MRAPI_SHMEM_ID_ANY`, then MRAPI will choose an internal id for you. All nodes in the nodes list must be initialized nodes in the system.

RETURN VALUE

On success a shared memory segment handle is returned, the address is filled in and *status is set to `MRAPI_SUCCESS`. On error, *status is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_SHMEM_ID_INVALID`	The `shmem_id` is not a valid shared memory segment id.
`MRAPI_ERR_SHM_NODES_INCOMPAT`	The list of nodes is not compatible for setting up shared memory.
`MRAPI_ERR_SHM_EXISTS`	This shared memory segment is already created.
`MRAPI_ERR_MEM_LIMIT`	No memory available.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized or one of the nodes in the list of nodes to share with is not initialized.
`MRAPI_ERR_PARAMETER`	Incorrect `size`, attributes, `attribute_size`, or `nodes_size` parameter.

NOTE

SEE ALSO

See `mrapi_shmem_init_attributes()` and `mrapi_shmem_set_attribute()`.

4.4.1.2 MRAPI_SHMEM_INIT_ATTRIBUTES

NAME

mrapi_shmem_init_attributes

SYNOPSIS

```
void mrapi_shmem_init_attributes(
  MRAPI_OUT mrapi_shmem_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Unless you want the defaults, this call must be used to initialize the values of an `mrapi_shmem_attributes_t` structure prior to `mrapi_shmem_set_attribute()`. You w would then use `mrapi_shmem_set_attribute()` to change any default values prior to calling `mrapi_shmem_create()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.1.3 MRAPI_SHMEM_SET_ATTRIBUTE

NAME

mrapi_shmem_set_attribute

SYNOPSIS

```
void mrapi_shmem_set_attribute(
  MRAPI_OUT mrapi_shmem_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_shmem_attributes_t` data structure prior to calling `mrapi_shmem_create()`. If the user wants to control which physical memory is used, then that is done by setting the MRAPI_SHMEM_RESOURCE attribute to the resource in the metadata tree. The user would first need to call `mrapi_resources_get`() and then iterate over the tree to find the desired resource (see the example use case for more details).

MRAPI pre-defined shared memory attributes:

Attribute num:	Description:	Datatype:	Default:
`MRAPI_SHMEM_RESOURCE`	The physical memory resource in the metadata resource tree that the memory should be allocated from.	`mrapi_resource_t`	`MRAPI_SHMEM_ANY`
`MRAPI_SHMEM_ADDRESS`	The requested address for a shared memory region	`mrapi_uint_t`	`MRAPI_SHMEM_ADDR_ANY`
`MRAPI_DOMAIN_SHARED`	Indicates whether or not this remote memory is shareable across domains.	`mrapi_boolean_t`	`MRAPI_TRUE`
`MRAPI_SHMEM_SIZE`	Returns the size of the shared memory segment in bytes. This attribute can only be set through the size parameter passed in to create.	`mrapi_size_t`	`No default.`

MRAPI_SHMEM_ADDRESS	if MRAPI_SHMEM_ANY then not necessarily contiguous, if <address> then contiguous;non-contiguous should be used with care and will not work in contexts that cannot handle virtual memory	mrapi_addr_t	MRAPI_SHMEM_ANY_CONTIGUOUS

RETURN VALUE

On success *status is set to MRAPI_SUCCESS. On error, *status is set to the appropriate error defined below.

ERRORS

MRAPI_ERR_ATTR_READONLY	Attribute can not be modified.
MRAPI_ERR_PARAMETER	Invalid attribute parameter.
MRAPI_ERR_ATTR_NUM	Unknown attribute number
MRAPI_ERR_ATTR_SIZE	Incorrect attribute size
MRAPI_ERR_NODE_NOTINIT	The calling node is not intialized.

NOTE

SEE ALSO

4.4.1.4 MRAPI_SHMEM_GET_ATTRIBUTE

NAME

mrapi_shmem_get_attribute

SYNOPSIS

```
void mrapi_shmem_get_attribute(
  MRAPI_IN mrapi_shmem_hndl_t shmem,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this shared memory. The attributes may be viewed but may not be changed (for this shared memory).

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MRAPI_ERR_SHM_INVALID`	Argument is not a valid shmem handle.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_shmem_set_attribute()` for a list of pre-defined attribute numbers.

4.4.1.5 MRAPI_SHMEM_GET

NAME

mrapi_shmem_get

SYNOPSIS

```
mrapi_shmem_hndl_t mrapi_shmem_get(
  MRAPI_IN mrapi_shmem_id_t shmem_id,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Given a `shmem_id` this function returns the MRAPI handle for referencing that shared memory segment.

RETURN VALUE

On success the shared memory segment handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_SHMEM_ID_INVALID`	The `shmem_id` is not a valid shared memory id or it was called with `shmem_id` set to `MRAPI_SHMEM_ID_ANY`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_SHM_NODE_NOTSHARED`	This shared memory is not shareable with the calling node. Which nodes it is shareable with was specified on the call to `mrapi_shmem_create()`.
`MRAPI_ERR_DOMAIN_NOTSHARED`	This resource can not be shared by this domain.

NOTE

Shared memory is the only MRAPI primitive that is always shareable across domains. Which nodes it is shared with is specified in the call to `mrapi_shmem_create()`.

SEE ALSO

4.4.1.6 MRAPI_SHMEM_ATTACH

NAME

mrapi_shmem_attach

SYNOPSIS

```
void* mrapi_shmem_attach(
  MRAPI_IN mrapi_shmem_hndl_t shmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attaches the caller to the shared memory segment and returns its address.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_SHM_INVALID`	Argument is not a valid shared memory segment handle.
`MRAPI_ERR_SHM_ATTACHED`	The calling node is already attached to the shared memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.1.7 MRAPI_SHMEM_DETACH

NAME

mrapi_shmem_detach

SYNOPSIS

```
void mrapi_shmem_detach(
  MRAPI_IN mrapi_shmem_hndl_t shmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function detaches the caller from the shared memory segment. All nodes must detach before any node can delete the memory.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_SHMEM_INVALID`	Argument is not a valid shared memory segment handle.
`MRAPI_ERR_SHM_NOTATTACHED`	The calling node is not attached to the shared memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.1.8 MRAPI_SHMEM_DELETE

NAME

mrapi_shmem_delete

SYNOPSIS

```
void mrapi_shmem_delete(
  MRAPI_IN mrapi_shmem_hndl_t shmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function deletes the shared memory segment if there are no nodes still attached to it. All nodes must detach before any node can delete the memory. Otherwise, delete will fail and there are no automatic retries nor deferred delete.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_SHM_INVALID`	Argument is not a valid shared memory segment handle.
`MRAPI_ERR_SHM_ATTCH`	There are nodes still attached to this shared memory segment thus it could not be deleted.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2 Remote Memory

The Remote Memory API's allow buffers in separate memory subsystems that are not directly accessible to be shared buffers. This can be accomplished if either CPU can see both memory regions, or if a DMA engine can provide a datapath to move the memory, or through some other form of communication that can perform the data transfer. These methods can optionally include a software cache.

If a CPU in the system can see both memory regions, then it can directly perform the memory transfers between memory spaces. A remote CPU node may not have access and must request the CPU that has access to perform any synchronization requests.

In a DMA transfer method, the DMA must have access to both memory regions. This entails set up of the buffer to be initially transferred between memory regions. The initial buffer and the copy are ready for access by either node. DMA can be used independently of a software cache or in conjunction with a software cache.

A software cache is similar to a hardware cache, and gives the ability to synchronize between different CPU's accessing the same memory structure which makes the accesses by both CPU's coherent. For example, when any write access is performed on a remote memory buffer, the result can be immediately stored in the software cache. If another CPU does a read or write access to the same region of the buffer, the software cache must communicate between CPU's and synchronize the buffer between remote memory regions prior to performing the buffer access. A sync command will force the remotely shared memory region to be synchronized.

4.4.2.1 MRAPI_RMEM_CREATE

NAME

mrapi_rmem_create

SYNOPSIS

```
mrapi_rmem_hndl_t mrapi_rmem_create(
   MRAPI_IN mrapi_rmem_id_t rmem_id,
   MRAPI_IN void* mem,
   MRAPI_IN mrapi_rmem_atype_t access_type,
   MRAPI_IN mrapi_rmem_attributes_t* attributes,
   MRAPI_IN mrapi_uint_t size,
   MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function promotes a private or shared memory segment on the calling node to a remote memory segment and returns a handle. The `mem` parameter is a pointer to the base address of the local memory buffer (see Section 3.5.2). Once a memory segment has been created, its attributes may not be changed. If the attributes are NULL, then implementation defined default attributes will be used. If `rmem_id` is set to `MRAPI_RMEM_ID_ANY`, then MRAPI will choose an internal id. `access_type` specifies access semantics. Access semantics are per remote memory buffer instance, and are either *strict* (meaning all clients must use the same access type), or *any* (meaning that clients may use any type supported by the MRAPI implementation). Implementations may define multiple access types (depending on underlying silicon capabilities), but must provide at minimum: `MRAPI_RMEM_ATYPE_ANY` (which indicates any semantics), and `MRAPI_RMEM_ATYPE_DEFAULT`, which has strict semantics Note that `MRAPI_RMEM_ATYPE_ANY` is only valid for remote memory buffer creation, clients must use `MRAPI_RMEM_ATYPE_DEFAULT` or another specific type of access mechanism provided by the MRAPI implementation (DMA, etc.) Specifying any type of access (even default) other than MRAPI_RMEM_ATYPE_ANY forces strict mode. The access type is explicitly passed in to create rather than being an attribute because it is so system specific, there is no easy way to define an attribute with a default value.

RETURN VALUE

On success a remote memory segment handle is returned, the address is filled in and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. In the case where the remote memory segment already exists, status will be set to `MRAPI_EXISTS` and the handle returned will not be a valid handle.

ERRORS

`MRAPI_ERR_RMEM_ID_INVALID`	The `rmem_id` is not a valid remote memory segment id.
`MRAPI_ERR_RMEM_EXISTS`	This remote memory segment is already created.
`MRAPI_ERR_MEM_LIMIT`	No memory available.
`MRAPI_ERR_RMEM_TYPENOTVALID`	Invalid `access_type` parameter
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_PARAMETER`	Incorrect `attributes`, `rmem`, or `size` parameter.
`MRAPI_ERR_RMEM_CONFLICT`	The memory pointer + size collides with another remote memory segment.

NOTE

This function is for promoting a segment of local memory (heap or stack, but stack would be dangerous and should be done with care) or an already created shared memory segment to rmem, but that also should be done with care.

SEE ALSO

See `mrapi_rmem_init_attributes()` and `mrapi_rmem_set_attribute()`.

See datatypes identifiers discussion: Section 3.12.13, access types: Section 3.5.2

4.4.2.2 MRAPI_RMEM_INIT_ATTRIBUTES

NAME

mrapi_rmem_init_attributes

SYNOPSIS

```
void mrapi_rmem_init_attributes(
  MRAPI_OUT mrapi_rmem_attributes_t* attributes,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Unless you want the defaults, this call must be used to initialize the values of an `mrapi_rmem_attributes_t` structure prior to `mrapi_rmem_set_attribute()`. You would then use `mrapi_rmem_set_attribute()` to change any default values prior to calling `mrapi_rmem_create()`.

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attributes` parameter
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.3 MRAPI_RMEM_SET_ATTRIBUTE

NAME

mrapi_rmem_set_attribute

SYNOPSIS

```
void mrapi_rmem_set_attribute(
  MRAPI_OUT mrapi_rmem_attributes_t* attributes,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function is used to change default values of an `mrapi_rmem_attributes_t` data structure prior to calling `mrapi_rmem_create()`.

MRAPI pre-defined remote memory attributes:

Attribute num:	Description:	Datatype:	Default:
`MRAPI_DOMAIN_SHARED`	Indicates whether or not this remote memory is shareable across domains.	`mrapi_boolean_t`	`MRAPI_TRUE`

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_ATTR_READONLY`	Attribute can not be modified.
`MRAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.4 MRAPI_RMEM_GET_ATTRIBUTE

NAME

mrapi_rmem_get_attribute

SYNOPSIS

```
void mrapi_rmem_get_attribute(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Returns the attribute that corresponds to the given `attribute_num` for this remote memory. The attributes may be viewed but may not be changed (for this remote memory).

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter.

ERRORS

`MRAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory handle.
`MRAPI_ERR_ATTR_NUM`	Unknown attribute number
`MRAPI_ERR_ATTR_SIZE`	Incorrect attribute size
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_rmem_set_attribute()` for a list of pre-defined attribute numbers.

4.4.2.5 MRAPI_RMEM_GET

NAME

mrapi_rmem_get

SYNOPSIS

```
mrapi_rmem_hndl_t mrapi_rmem_get(
   MRAPI_IN mrapi_rmem_id_t rmem_id,
   MRAPI_IN mrapi_rmem_atype_t access_type,
   MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

Given a `rmem_id`, this function returns the MRAPI handle referencing to that remote memory segment. `access_type` specifies access semantics. Access semantics are per remote memory buffer instance, and are either *strict* (meaning all clients must use the same access type), or any (meaning that clients may use any type supported by the MRAPI implementation). Implementations may define multiple access types (depending on underlying silicon capabilities), but must provide at minimum: `MRAPI_RMEM_ATYPE_ANY` (which indicates any semantics), and `MRAPI_RMEM_ATYPE_DEFAULT`, which has strict semantics Note that `MRAPI_RMEM_ATYPE_ANY` is only valid for remote memory buffer creation, clients must use `MRAPI_RMEM_ATYPE_DEFAULT` or another specific type of access mechanism provided by the MRAPI implementation (DMA, etc.) The access type must match the access type that the memory was created with unless the memory was created with the `MRAPI_RMEM_ATYPE_ANY` type. See Section 3.5.2 for a discussion of remote memory access types.

RETURN VALUE

On success the remote memory segment handle is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_ID_INVALID`	The `rmem_id` parameter does not refer to a valid remote memory segment or it was called with `rmem_id` set to `MRAPI_RMEM_ID_ANY`.
`MRAPI_ERR_RMEM_ATYPE_INVALID`	Invalid `access_type` parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MRAPI_ERR_DOMAIN_NOTSHARED`	This resource can not be shared by this domain.
`MRAPI_ERR_RMEM_ATYPE`	Type specified on attach is incompatible with type specified on create.

NOTE

SEE ALSO

`mrapi_rmem_set_attribute()`, access types: Section 3.5.2

4.4.2.6 MRAPI_RMEM_ATTACH

NAME

mrapi_rmem_attach

SYNOPSIS

```
void mrapi_rmem_attach(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function attaches the caller to the remote memory segment. Once this is done, the caller may use the `mrapi_rmem_read()` and `mrapi_rmem_write()` functions. The caller should call `mrapi_rmem_detach()` when finished using the remote memory.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_ATTACHED`	The calling node is already attached to the remote memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

Section 3.5.2

4.4.2.7 MRAPI_RMEM_DETACH

NAME

mrapi_rmem_detach

SYNOPSIS

```
void mrapi_rmem_detach(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function detaches the caller from the remote memory segment. All attached nodes must detach before any node can delete the memory.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.8 MRAPI_RMEM_DELETE

NAME

mrapi_rmem_delete

SYNOPSIS

```
void mrapi_rmem_delete(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function demotes the remote memory segment. All attached nodes must detach before the node can delete the memory. Otherwise, delete will fail and there are no automatic retries nor deferred delete. Note that memory is not de-allocated it is just no longer accessible via the MRAPI remote memory function calls. Only the node that created the remote memory can delete it.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_ATTACH`	Unable to demote the remote memory because other nodes are still attached to it.
`MRAPI_ERR_RMEM_NOTOWNER`	The calling node is not the one that created the remote memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.9 MRAPI_RMEM_READ

NAME

mrapi_rmem_read

SYNOPSIS

```
void mrapi_rmem_read(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_IN mrapi_uint32_t rmem_offset,
  MRAPI_OUT void* local_buf,
  MRAPI_IN size_t local_buf_size,
  MRAPI_IN mrapi_uint32_t local_offset,
  MRAPI_IN mrapi_uint32_t bytes_per_access,
  MRAPI_IN mrapi_uint32_t num_strides,
  MRAPI_IN mrapi_uint32_t rmem_stride,
  MRAPI_IN mrapi_uint32_t local_stride,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function performs `num_strides` memory reads, where each read is of size `bytes_per_access` bytes. The *i*-th read copies `bytes_per_access` bytes of data from `rmem` with offset `rmem_offset + i*rmem_stride` to `local_buf` with offset `local_offset + i*local_stride`, where 0 <= *i* < `num_strides`. The local_buf_size represents the number of bytes in the `local_buf`.

This supports scatter/gather type accesses. To perform a single read, without the need for scatter/gather, set the `num_strides` parameter to 1.

This routine blocks until memory can be read.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_BUFF_OVERRUN`	`rmem_offset + (rmem_stride * num_strides )` would fall out of bounds of the remote memory buffer.
`MRAPI_ERR_RMEM_STRIDE`	`num_strides>1` and `rmem_stride` and/or `local_stride` are less than `bytes_per_access.`
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_PARAMETER`	Either the `local_buf` is invalid or the `buf_size` is zero or `bytes_per_access` is zero.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.10 MRAPI_RMEM_READ_I

NAME

mrapi_rmem_read_i

SYNOPSIS

```
void mrapi_rmem_read_i(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_IN mrapi_uint32_t rmem_offset,
  MRAPI_OUT void* local_buf,
  MRAPI_IN mrapi_uint32_t local_offset,
  MRAPI_IN mrapi_uint32_t bytes_per_access,
  MRAPI_IN mrapi_uint32_t num_strides,
  MRAPI_IN mrapi_uint32_t rmem_stride,
  MRAPI_IN mrapi_uint32_t local_stride,
  MRAPI_OUT mrapi_request_t* mrapi_request,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This (non-blocking) function performs `num_strides` memory reads, where each read is of size `bytes_per_access` bytes. The *i*-th read copies `bytes_per_access` bytes of data from `rmem` with offset `rmem_offset + i*rmem_stride` to `local_buf` with offset `local_offset + i*local_stride`, where 0 <= *i* < `num_strides`.

This supports scatter/gather type accesses. To perform a single read, without the need for scatter/gather, set the `num_strides` parameter to 1.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. Use `mrapi_test()`, `mrapi_wait()` or `mrapi_wait_any()` to test for completion of the operation.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_BUFF_OVERRUN`	`rmem_offset + (rmem_stride * num_strides )` would fall out of bounds of the remote memory buffer.
`MRAPI_ERR_RMEM_STRIDE`	`num_strides`>1 and `rmem_stride` and/or `local_stride` are less than `bytes_per_access`.
`MRAPI_ERR_REQUEST_LIMIT`	`No more request handles available.`
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_RMEM_BLOCKED`	We have hit a hardware limit of the number of asynchronous DMA/cache operations that can be pending ("in flight") simultaneously. Thus we now have to block because the resource is busy.
`MRAPI_ERR_PARAMETER`	Either the `local_buf` is invalid or the `buf_size` is zero or `bytes_per_access` is zero.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.11 MRAPI_RMEM_WRITE

NAME

mrapi_rmem_write

SYNOPSIS

```
void mrapi_rmem_write(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_IN mrapi_uint32_t rmem_offset,
  MRAPI_IN void* local_buf,
  MRAPI_IN mrapi_uint32_t local_offset,
  MRAPI_IN mrapi_uint32_t bytes_per_access,
  MRAPI_IN mrapi_uint32_t num_strides,
  MRAPI_IN mrapi_uint32_t rmem_stride,
  MRAPI_IN mrapi_uint32_t local_stride,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This function performs `num_strides` memory writes, where each write is of size `bytes_per_access` bytes. The *i*-th write copies `bytes_per_access` bytes of data from `local_buf` with offset `local_offset + i*local_stride` to `rmem` with offset `rmem_offset + i*rmem_stride`, where 0 <= *i* < `num_strides`.

This supports scatter/gather type accesses. To perform a single write, without the need for scatter/gather, set the `num_strides` parameter to 1.

This routine blocks until memory can be written.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_BUFF_OVERRUN`	`rmem_offset + (rmem_stride * num_strides )` would fall out of bounds of the remote memory buffer.
`MRAPI_ERR_RMEM_STRIDE`	`num_strides`>1 and `rmem_stride` and/or `local_stride` are less than `bytes_per_access.`
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_PARAMETER`	Either the `local_buf` is invalid or `bytes_per_access` is zero.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.12 MRAPI_RMEM_WRITE_I

NAME

mrapi_rmem_write_i

SYNOPSIS

```
void mrapi_rmem_write_i(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_IN mrapi_uint32_t rmem_offset,
  MRAPI_IN void* local_buf,
  MRAPI_IN mrapi_uint32_t local_offset,
  MRAPI_IN mrapi_uint32_t bytes_per_access,
  MRAPI_IN mrapi_uint32_t num_strides,
  MRAPI_IN mrapi_uint32_t rmem_stride,
  MRAPI_IN mrapi_uint32_t local_stride,
  MRAPI_OUT mrapi_request_t* mrapi_request,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

This (non-blocking) function performs `num_strides` memory writes, where each write is of size `bytes_per_access` bytes. The *i*-th write copies `bytes_per_access` bytes of data from `local_buf` with offset `local_offset + i*local_stride` to `rmem` with offset `rmem_offset + i*rmem_stride`, where 0 <= *i* < `num_strides`.

This supports scatter/gather type accesses. To perform a single write, without the need for scatter/gather, set the `num_strides` parameter to 1.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. Use `mrapi_test()`, `mrapi_wait()` or `mrapi_wait_any()` to test for completion of the operation.

ERRORS

`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_BUFF_OVERRUN`	`rmem_offset + (rmem_stride * num_strides )` would fall out of bounds of the remote memory buffer.
`MRAPI_ERR_RMEM_STRIDE`	`num_strides`>1 and `rmem_stride` and/or `local_stride` are less than `bytes_per_access`.
`MRAPI_ERR_REQUEST_LIMIT`	`No more request handles available.`
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_RMEM_BLOCKED`	We have hit a hardware limit of the number of asynchronous DMA/cache operations that can be pending ("in flight") simultaneously. Thus we now have to block because the resource is busy.
`MRAPI_ERR_PARAMETER`	Either the `local_buf` is invalid or `bytes_per_access` is zero.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.13 MRAPI_RMEM_FLUSH

NAME

mrapi_rmem_flush

SYNOPSIS

```
void mrapi_rmem_flush(
  MRAPI_IN mrapi_rmem_hndl_t rmem,
  MRAPI_OUT mrapi_status_t* status
  );
```

DESCRIPTION

This function flushes the remote memory. Support for this function is optional and on some systems this may not be supportable. However, if an implementation wants to support coherency back to main backing store then this is the way to do it. Note, that this is not an automatic synch back to other viewers of the remote data and they would need to also perform a synch, so it is 'application managed' coherency. If writes are synchronizing, then a flush will be a no-op.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_NOT_SUPPORTED`	The flush call is not supported
`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.4.2.14 MRAPI_RMEM_SYNC

NAME

mrapi_rmem_sync

SYNOPSIS

```
void mrapi_rmem_sync(
  MRAPI_IN mrapi_rmem_handle_t rmem,
  MRAPI_OUT mrapi_status_t* status
  );
```

DESCRIPTION

This function synchronizes the remote memory. This function provides application managed coherency. It does not guarantee that all clients of the rmem buffer will see the updates, see corresponding `mrapi_rmem_flush()`. For some underlying hardware this may not be possible. MRAPI implementation can return an error if the synch cannot be performed.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_NOT_SUPPORTED`	The synch call is not supported
`MRAPI_ERR_RMEM_INVALID`	Argument is not a valid remote memory segment handle.
`MRAPI_ERR_RMEM_NOTATTACHED`	The caller is not attached to the remote memory.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.5 MRAPI non-blocking operations

The MRAPI provides both blocking and non-blocking versions of communication functions that may be delayed because the implementation requires synchronization between multiple nodes. The non-blocking version of functions is denoted by an _i() suffix. For example, the `mrapi_rmem_write()` function copies a data buffer from local memory to a remote shared memory buffer. Since the data copy operation might take many cycles, MRAPI also provides `mrapi_rmem_write_i()` function, which initiates the DMA operation and returns immediately. Like all non-blocking functions, `mrapi_rmem_write_i()` fills in a `mrapi_request_t` object before returning.

The `mrapi_request_t` object provides a unique identifier for each in-flight non-blocking operation. These 'request handles' can be passed to the `mrapi_test(), mrapi_wait()`, or `mrapi_wait_any()` methods in order to find out when the non-blocking operation has completed. When one of these API calls determines that a non-blocking request has finished, it returns indicating completion and fills in an `mrapi_status_t` object to indicate why the request completed. The status object contains an error code indicating whether the operation finished successfully or was terminated because of an error. The `mrapi_request_t` is an opaque data type and the user should not attempt to examine it.

Non-blocking operations may consume system resources until the programmer confirms completion by calling `mrapi_test(), mrapi_wait(),` or `mrapi_wait_any()`. Thus, the programmer should be sure to confirm completion of every non-blocking operation via these APIs. Alternatively, an in-flight operation can be cancelled by calling `mrapi_cancel()`. This function forces the operations specified by the `mrapi_request_t` object to stop immediately, releasing any system resources allocated in order to perform the operation.

4.5.1 MRAPI_TEST

NAME

mrapi_test

SYNOPSIS

```
mrapi_boolean_t mrapi_test(
  MRAPI_IN mrapi_request_t* request,
  MRAPI_OUT size_t* size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_test()` 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 size parameter is not currently used but is there to align with MCAPI.

RETURN VALUE

On success, `MRAPI_TRUE` is returned and *status is set to `MRAPI_SUCCESS`. If the operation has not completed `MRAPI_FALSE` is returned and *status is set to `MRAPI_INCOMPLETE`. On error, `MRAPI_FALSE` is returned and *status is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle.
`MRAPI_ERR_REQUEST_CANCELED`	The request was canceled.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.5.2 MRAPI_WAIT

NAME

mrapi_wait

SYNOPSIS

```
mrapi_boolean_t mrapi_wait(
  MRAPI_IN mrapi_request_t* request,
  MRAPI_OUT size_t* size,
  MRAPI_IN mrapi_timeout_t timeout,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_wait()` waits 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 size parameter is not currently used but is there to align with MCAPI.

RETURN VALUE

On success `MRAPI_TRUE` is returned and status is set to `MRAPI_SUCCESS`. On error `MRAPI_FALSE` is returned and `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle.
`MRAPI_ERR_REQUEST_CANCELED`	The request was canceled.
`MRAPI_TIMEOUT`	The operation timed out.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

SEE ALSO

4.5.3 MRAPI_WAIT_ANY

NAME

mrapi_wait_any

SYNOPSIS

```
mrapi_uint_t mrapi_wait_any(
  MRAPI_IN size_t num_requests,
  MRAPI_IN mrapi_request_t*  requests,
  MRAPI_OUT size_t* size,
  MRAPI_IN mrapi_timeout_t timeout ,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_wait_any()` waits 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 operation has completed. If more than one request has completed, it will return the first one it finds. number is the number of requests in the array. requests is the array of `mrapi_request_t` identifiers for the non-blocking operations. The size parameter is not currently used but is there to align with MCAPI.

RETURN VALUE

On success, the index into the requests array of the `mrapi_request_t` identifier that has completed or has been canceled is returned and `*status` is set to `MRAPI_SUCCESS`. On error, `-1` is returned and `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle.
`MRAPI_ERR_REQUEST_CANCELED`	The request was canceled.
`MRAPI_TIMEOUT`	The operation timed out.
`MRAPI_ERR_PARAMETER`	Incorrect number (if=0) requests parameter.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.5.4 MRAPI_CANCEL

NAME

mrapi_cancel

SYNOPSIS

```
void mrapi_cancel(
  MRAPI_IN mrapi_request_t* request,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_cancel()` cancels an outstanding request. Any pending calls to `mrapi_wait()` or `mrapi_wait_any()` for this request will also be cancelled. The returned `status` of a canceled `mrapi_wait()` or `mrapi_wait_any()` call will indicate that the request was cancelled. Only the node that initiated the request may call cancel.

RETURN VALUE

On success, *`status` is set to MRAPI_SUCCESS. On error, *`status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_REQUEST_INVALID`	Argument is not a valid request handle for this node.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.6 MRAPI Metadata

MRAPI supports the searching and querying of metadata about the host system. The host system has a set of resources, with each resource having a set of attributes. Each attribute has a value.

A central concept of the MRAPI metadata support is the data structure that represents resources in a system. A call to `mrapi_resources_get()` will result in the creation of a data structure in the form of a tree. The nodes are the resources, and the edges represent scope (not ownership). By navigating the data structure, the user can locate the resource desired, and then use the `mrapi_resource_get_attribute()` function to obtains the value of an attribute. The function `mrapi_resource_tree_free()` is used to free the memory used by the data structure. A node can only see the resources in it's domain and a given domain's scope may change over time if the system is repartitioned for power, hypervisor, etc.

The source for the MRAPI metadata system resources can be initialized in several ways. Each implementation upon initialization can obtain resource information from a number of ways, including from standard information systems like SPIRIT Consortium's IP-XACT and Linux device trees.

The MRAPI metadata supports dynamic attributes (attributes with values that change in time). MRAPI supports the ability to start, stop, reset, and query dynamic attributes. Dynamics attributes are not required to be supported.

MRAPI also supports registering callbacks that are called when an event occurs. Events can include an attribute exceeding a threshold, or a counter rollover. Callbacks are not required to be supported when no events are defined by the implementation.

4.6.1 MRAPI_RESOURCES_GET

NAME

mrapi_resources_get

SYNOPSIS

```
mrapi_resource_t* mrapi_resources_get(
  MRAPI_IN mrapi_rsrc_filter_t subsystem_filter,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_resources_get()` returns a tree of system resources available to the calling node, at the point in time when it is called (this is dynamic in nature). `mrapi_resource_get_attribute()` can be used to make a specific query of an attribute of a specific system resource. `subsystem_filter` is an enumerated type that is used as a filter indicating the scope of the desired information MRAPI returns. See Section 3.6.1 for a description of how to navigate the resource tree as well as section 6.1 for an example use case.

The valid subsystem filters are:

`MRAPI_RSRC_MEM, MRAPI_RSRC_CACHE, MRAPI_RSRC_CPU`

RETURN VALUE

On success, returns a pointer to the root of a tree structure containing the available system resources, and `*status` is set to `MRAPI_SUCCESS`. On error, `MRAPI_NULL` is returned and `*status` is set to the appropriate error defined below. The memory associated with the data structures returned by this function is system managed and must be released via a call to `mrapi_resource_tree_free()`.

ERRORS

`MRAPI_ERR_RSRC_INVALID_SUBSYSTEM`	Argument is not a valid `subsystem_filter` value.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_resource_get_attribute()`, Section 3.6.1 and Section 3.12.4

4.6.2 MRAPI_RESOURCE_GET_ATTRIBUTE

NAME

mrapi_resource_get_attribute

SYNOPSIS

```
void mrapi_resource_get_attribute(
  MRAPI_IN mrapi_resource_t* resource,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT void* attribute,
  MRAPI_IN size_t attribute_size,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_resource_get_attribute()` returns the attribute value at the point in time when this function is called (the value of an attribute may be dynamic in nature), given the input resource and attribute number. `resource` is a pointer to the respective resource, `attribute_num` is the number of the attribute to query for that resource, and `attribute_size` is the size of the attribute. Resource attributes are read-only. Attribute numbers are assigned by the MRAPI implementation and are specific to the given resource type (see Section 3.6.1).

The tables below show the valid `attribute_nums` for each type of resource:

type of `mrapi_resource_t` = `MRAPI_RSRC_MEM`

attribute_num:	datatype:
MRAPI_RSRC_MEM_BASEADDR	mrapi_addr_t
MRAPI_RSRC_MEM_WORDSIZE	mrapi_uint_t
MRAPI_RSRC_MEM_NUMWORDS	mrapi_uint_t

type of `mrapi_resource_t` = `MRAPI_RSRC_CACHE`

attribute_num:	datatype:
MRAPI_RSRC_CACHE_SIZE	mrapi_uint_t
MRAPI_RSRC_CACHE_LINE_SIZE	mrapi_uint_t
MRAPI_RSRC_CACHE_ASSOCIATIVITY	mrapi_uint_t
MRAPI_RSRC_CACHE_LEVEL	mrapi_uint_t

type of `mrapi_resource_t` = `MRAPI_RSRC_CPU`

attribute_num:	datatype:
MRAPI_RSRC_CPU_FREQUENCY	mrapi_uint_t
MRAPI_RSRC_CPU_TYPE	char*
MRAPI_RSRC_CPU_ID	mrapi_uint_t

RETURN VALUE

On success `*status` is set to `MRAPI_SUCCESS` and the attribute value is filled in. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. The attribute identified by the `attribute_num` is returned in the `void* attribute` parameter.

ERRORS

MRAPI_ERR_RSRC_INVALID	Invalid resource
MRAPI_ERR_ATTR_NUM	Unknown attribute number
MRAPI_ERR_ATTR_SIZE	Incorrect attribute size
MRAPI_ERR_PARAMETER	Invalid attribute parameter.
MRAPI_ERR_NODE_NOTINIT	The calling node is not intialized.

NOTE

SEE ALSO

```
mrapi_resources_get()
```

4.6.3 MRAPI_DYNAMIC_ATTRIBUTE_START

NAME

mrapi_dynamic_attribute_start

SYNOPSIS

```
void mrapi_dynamic_attribute_start(
  MRAPI_IN mrapi_resource_t* resource,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_IN void (*rollover_callback) (void),
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_dynamic_attribute_start()` sets the system up to begin collection of the attribute's value over time. `resource` is a pointer to the given resource, `attribute_num` is the number of the attribute to start monitoring for that resource. Attribute numbers are specific to the given resource type.

The `rollover_callback` is an optional function pointer. If supplied the implementation will call the function when the specified attribute value rolls over from its maximum value. If this callback is not supplied the attribute will roll over silently.

If you call stop and then start again, the resource will start at it's previous value. To reset it, call `mrapi_dynamic_attribute_reset()`.

RETURN VALUE

On success, `*status` is set to `MRAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RSRC_INVALID`	Invalid resource
`MRAPI_ERR_ATTR_NUM`	Invalid attribute number
`MRAPI_ERR_RSRC_NOTDYNAMIC`	The input attribute is static and not dynamic, and therefore can't be started.
`MRAPI_ERR_RSRC_STARTED`	The attribute is dynamic and has already been started
`MRAPI_ERR_RSRC_COUNTER_INUSE`	The counter is currently in use by another node.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_dynamic_attribute_stop()`, Section 3.12.4

4.6.4 MRAPI_DYNAMIC_ATTRIBUTE_RESET

NAME

mrapi_dynamic_attribute_reset

SYNOPSIS

```
void mrapi_dynamic_attribute_reset(
  MRAPI_IN mrapi_resource_t *resource,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_dynamic_attribute_reset()` resets the value of the collected dynamic attribute. resource is the given resource, `attribute_num` is the number of the attribute to reset. Attribute numbers are specific to a given resource type.

RETURN VALUE

On success, `*status` is set to MRAPI_SUCCESS. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RSRC_INVALID`	Invalid resource
`MRAPI_ERR_ATTR_NUM`	Invalid attribute number
`MRAPI_ERR_RSRC_NOTDYNAMIC`	The input attribute is static and not dynamic, and therefore can't be reset.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

Some dynamic attributes do not have a defined reset value. In this case, calling `mrapi_dynamic_attribute_reset()` has no effect.

SEE ALSO

Section 3.6.1

4.6.5 MRAPI_DYNAMIC_ATTRIBUTE_STOP

NAME

mrapi_dynamic_attribute_stop

SYNOPSIS

```
void mrapi_dynamic_attribute_stop(
  MRAPI_IN mrapi_resource_t* resource,
  MRAPI_IN mrapi_uint_t attribute_num,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_dynamic_attribute_stop()` stops the system from collecting dynamic attribute values. `resource` is the given resource, `attribute_num` is the number of the attribute to stop monitoring. Attribute numbers are specific to a given resource type. If you call stop and then start again, the resource will start at it's previous value. To reset it, call `mrapi_dynamic_attribute_reset()`.

RETURN VALUE

On success, `*status` is set to MRAPI_SUCCESS. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RSRC_INVALID`	Invalid resource
`MRAPI_ERR_ATTR_NUM`	Invalid attribute number
`MRAPI_ERR_RSRC_NOTDYNAMIC`	The input attribute is static and not dynamic, and therefore can't be stopped.
`MRAPI_ERR_RSRC_NOTSTARTED`	The attribute is dynamic and has not been started by the calling node.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

`mrapi_dynamic_attribute_start()`

4.6.6 MRAPI_RESOURCE_ REGISTER_CALLBACK

NAME

mrapi_resource_ register_callback

SYNOPSIS

```
void mrapi_resource_register_callback(
  MRAPI_IN mrapi_event_t event,
  MRAPI_IN unsigned int frequency,
  MRAPI_IN void (*callback_function) (mrapi_event_t event),
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_register_callback()` registers an application-defined function to be called when a specific system event occurs. The set of available events is implementation defined. Some implementations may choose not to define any events and thus not to support this functionality. The `frequency` parameter is used to indicate the reporting frequency for which an event should trigger the callback (frequency is specified in terms of number of event occurrences, e.g., callback on every nth occurrence where n=frequency). An example usage of `mrapi_register_callback()` could be for notification when the core experiences a power management event so that the application can determine the cause (manual or automatic) and/or the level (nap, sleep, or doze, etc.), and use this information to adjust resource usages.

RETURN VALUE

On success, the `callback_function()` will be registered for the event, and `*status` is set to MRAPI_SUCCESS. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RSRC_INVALID_EVENT`	Invalid event
`MRAPI_ERR_RSRC_INVALID_CALLBACK`	Invalid callback function
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

SEE ALSO

4.6.7 MRAPI_RESOURCE_TREE_FREE

NAME

mrapi_resource_tree_free

SYNOPSIS

```
void mrapi_resource_tree_free(
  mrapi_resource_t* MRAPI_IN* root,
  MRAPI_OUT mrapi_status_t* status
);
```

DESCRIPTION

`mrapi_resource_tree_free()` frees the memory in a resource tree. `root` is the root of a resource tree originally obtained from a call to `mrapi_resources_get()`.

RETURN VALUE

On success, `*status` is set to MRAPI_SUCCESS and root will be set to `MRAPI_NULL`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MRAPI_ERR_RSRC_INVALID_TREE`	Invalid resource tree
`MRAPI_ERR_RSRC_NOTOWNER`	The calling node is not the same node that originally called `mrapi_resources_get()`.
`MRAPI_ERR_NODE_NOTINIT`	The calling node is not intialized.

NOTE

Subsequent usage of `root` will give undefined results.

SEE ALSO

`mrapi_resources_get()`

4.7 MRAPI Convenience Functions

MRAPI supports a convenience function for displaying the status parameter.

4.7.1 MRAPI_DISPLAY_STATUS

NAME

mrapi_display_status

SYNOPSIS

```
char* mrapi_display_status(
   MRAPI_IN mrapi_status_t mrapi_status,
   MRAPI_OUT char* status_message,
  MRAPI_IN size_t size
);
```

DESCRIPTION

`mrapi_display_status()` displays the status parameter as a string by copying it into the user supplied buffer: `status_message`.

RETURN VALUE

mrapi_true is returned on success, otherwise mrapi_false is returned. If the status is an unknown status, `status_message` will be set to "UNKNOWN".

ERRORS

NONE DEFINED	N/A

NOTE

SEE ALSO

5 FAQ

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

A: A reference implementation is planned in the future. The current plan is to receive feedback on the draft specification and make modifications based upon the feedback. When the specification is near finalization, the MRAPI working group will announce the plans and schedule for such an implementation. The reference implementation models the functionality of the specification and does not intend to be a high performance implementation.

Q: Can you elaborate on how a hardware accelerators will interact with embedded processors using MRAPI? An API is a library of C/C++ functions, but it is not clear to me how an API can be used with a hardware accelerator, which can be very application specific.

A: The API can be implemented on top of a hardware accelerator. For example, an SoC may have hardware acceleration for mutexes, in which case an MRAPI implementation could utilize that hardware accelerator without the programmer needing to know how to interact with it directly.

Q: Does the API have testcases?

A: The API itself does not have testcases. However, as with the MCAPI example implementation which is available from the Multicore Association we would expect an MRAPI example implementation to contain testcases.

Q: Do you have implementations of the API that can be tested by the reviewers?

A: We are hoping to publish a simple POSIX implementation along with the spec.

Q: I assume MRAPI relies upon a "local" resource manager ? That is MRAPI must store state, and so needs a way to allocate state storage ?

A: It is up to the MRAPI implementation as to how resources are managed. Our simple initial implementation stores state in shared memory protected with a semaphore.

Q: I saw a statement that other solutions are too heavyweight because they target distributed systems. Does it mean that your goal is not to target the distributed system? What happens when we have a multi-chip multi-core? Isn't this the same distributed system?

A: To be clear, MRAPI targets cores on a chip, and chips on a board. MRAPI is not intended to scale beyond that scope.

Q: Is it possible to hide the differences between local and remote memory by just different properties of these memories? Remote memory will have higher latency, some access restrictions, etc.

A: The working group has considered the possibility of allowing the “promotion” of local memory to remote memory, and then allowing all memory accesses to occur through the API. This would effectively hide the difference, but at a performance cost. For now this is a deferred feature.

Q: In many HW systems transitions between low power (or no power) and fully working conditions are extremely frequent. In such systems some state change callbacks will become a nightmare. How are you planning to handle the situation?

A: In the situation where the application does not want to be disturbed by frequent callbacks then it would be better for the application to periodically poll MRAPI at a time of its own choosing. This is certainly possible with MRAPI.

Q: What is the idea of API asking for HW accelerators if these accelerators are actually powered off because of inactivity?

A: In such a scenario the application would determine that there was no acceleration available and would have to find an alternative means to perform its work, perhaps by executing code on the CPU.

Q: Are there any plans to include trigger APIs? For example, invoke callback when a particular resource hits some pre-defined conditions/threshold?

A: Currently there are no threshold-related callbacks other than counter wrap-arounds. MRAPI may consider this for a future version.

Q: Primitives - didn't you consider including RCUs (Read Copy Update locks)?

A: The MRAPI working group did consider read copy update locks. After discussion with some of the original creators of the RCU code for Linux we determined that for now there is not sufficient evidence that a high performance user-level implementation of RCU was feasible. We intend to monitor developments in this area as we are aware that it is an active area of research.

Q: These primitives are necessary, but seem to be insufficient. I would think that the goal of MRAPI would include the ability to write a resource manager that any app that uses MRAPI could plug into. That implies that at a minimum: resource enumerations should be standardized, or a mechanism for self-describing enumerations be created.

A: MRAPI is intended to provide some of the primitives that could be used for creating a higher level resource manager. However, it is also intended to be useful for application level programmers to write multicore code, and for this reason it was kept minimal and orthogonal to other Multicore Association APIs. The working group believes that a full-featured resource manager would require all of the Multicore Association APIs, e.g., MCAPI, MRAPI, and MTAPI.

Q: Are any companies currently incorporating or have plans to incorporate MRAPI in their products. If so, can you name the products?

A: At this time there have been no public announcements. There is at least one university research project that is looking at MRAPI for heterogeneous multicore computing. We expect more activities to emerge once the spec is released officially.

Q: Is MRAPI planned to be processor agnostic?

A: Yes, that is the plan.

Q: Is MRAPI dependent on any other Resource Management standards and/or approaches?

A: No, there should be no such dependencies in MRAPI.

Comment [JH1]: Need to update these, and add the metadata use cases

6 Use Cases

6.1 Simple example of creating shared memory using metadata

This use case illustrates how a user would control which physical memory shared memory is allocated from by walking a filtered resource tree and selecting a particular memory resource. The default is to allow the system to control where shared memory is allocated from.

```
mrapi_status_t status;
mrapi_resource_t* mem_root;
mrapi_shmem_hndl_t shmem_hndl;
mrapi_shmem_attributes_t shmem_attributes;
int i;
mrapi_addr_t addr;

// get the metadata resource tree (filtered for memory only)
mem_root = mrapi_resources_get (MRAPI_RSRC_MEM,&status);
if (status != MRAPI_SUCCESS) { ERR("Unable to get resource tree");}

// find the desired memory in the metadata resource tree
for (i = 0; i < mem_root->child_count; i++) {
            mrapi_resource_get_attribute (
                  mem_root->children[i],
                  MRAPI_RSRC_MEM_BASEADDR,
                  &addr,
                  sizeof(mrapi_addr_t),
                  &status);
      if (status != MRAPI_SUCCESS) { ERR ("Unable to get resource attr");}

      if (addr == 0xfffff000) {
         // we've found the resource for the region we want to use

         // set up the shared memory resource attribute with the metadata
         mrapi_shmem_init_attributes (&shmem_attributes, &status);
         if (status != MRAPI_SUCCESS) { ERR ("Unable to init shmem attrs");}

         mrapi_shmem_set_attribute (&shmem_attributes,
                        MRAPI_SHMEM_RESOURCE,
                        mem_root->children[i],
                        sizeof(mrapi_resource_t),
                        &status);
         if (status != MRAPI_SUCCESS) { ERR("Unable to set shmem attrs");}

        // create the shared memory
        shmem_hndl = mrapi_shmem_create (MRAPI_SHMEM_ID_ANY,
                        1024 /* size */,
                        NULL /*share with all nodes*/,
                        0 /*nodes_size*/,
```

```
                        &shmem_attributes,
                        sizeof(shmem_attributes),
                        &status);

          if (status != MRAPI_SUCCESS) { ERR("Unable to create shmem");}
          break;
       }
    }
}
```

6.2 Automotive Use Case

6.2.1 Characteristics

6.2.1.1 Sensors

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

6.2.1.2 Control Task

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

6.2.1.3 Lost Data

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

6.2.1.4 Types of Tasks

Consists of both control and signal processing, especially FFT.

6.2.1.5 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.

6.2.1.6 Message Size and Frequency

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

6.2.1.7 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.

6.2.1.8 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 small OS or no OS is involved it is typical for logical mappings of addresses to be avoided, where 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 is to be avoided.

6.2.2 Key functionality requirements

6.2.2.1 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.

6.2.2.2 Angle Task

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

6.2.2.3 Data Tasks

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

6.2.3 Context/Constraints

6.2.3.1 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.

6.2.3.2 Polling/Interrupts

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

6.2.3.3 Reliability

Sensors assumed to be reliable; interconnect 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.

6.2.4 Metrics

6.2.4.1 Latency of the control task

Dependent on engine RPM. At 1800 RPM the task must complete every 33.33ms, and at 9000 RPM the task must complete every 6.667ms.

6.2.4.2 # dropped sensor readings

Ideally zero.

6.2.4.3 Latencies of data tasks

Ideally the sum of the latencies plus message send/receive times should be < latency of the control loop given 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.

6.2.4.4 Code size

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

6.2.5 Possible Factorings

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

6.2.6 MRAPI 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 uni-core or multi-core
- 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)

6.2.7 Mental Models

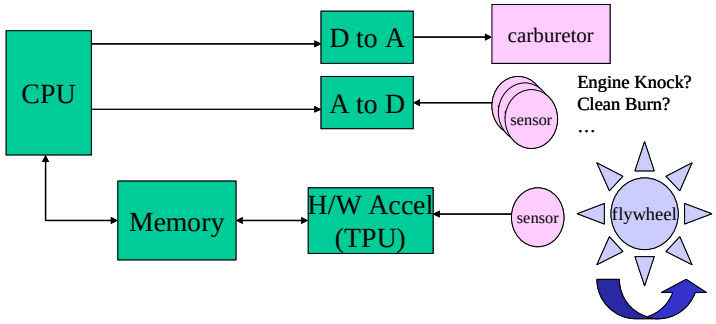

Figure 1 – example hardware

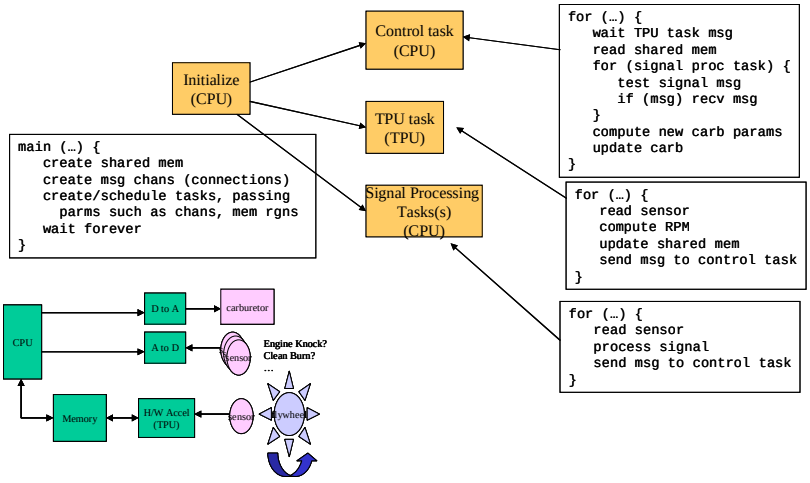

Figure 2 – A possible mapping

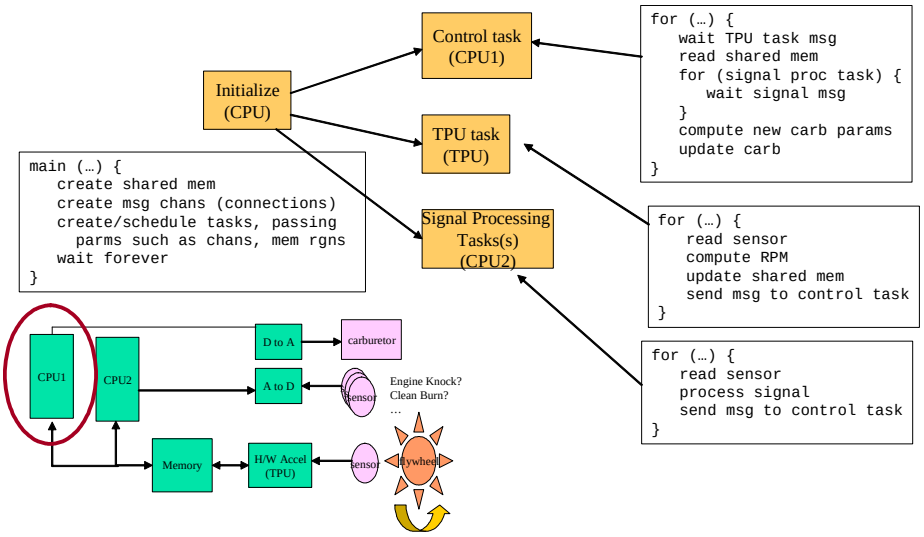

Figure 3 – Alternative Hardware

6.2.8 MRAPI pseudo-code

6.2.8.1 Initial Mapping

```
////////////////////////////////////////
// The control task
////////////////////////////////////////
void Control_Task(void) {
   mrapi_shmem_hndl_t sMem;   /* handle to the shmem */
   mrapi_mutex_hndl_t sMem_mutex;
   char* sPtr;
   mrapi_key_t lock_key;
   uint8_t tFlag;
   mcapi_endpoint_t tpu_rmem_endpt;
   mcapi_endpoint_t sig_endpt, sig_rmem_endpt;
   mcapi_endpoint_t tmp_endpt;
   mcapi_pktchan_recv_hndl_t sig_chan;
   struct SIG_DATA sDat;
   size_t tSize;
   mcapi_request_t r1, r2;
   mcapi_status_t err;
   mrapi_status_t mrapi_status;
  mrapi_parameters_t parms;
  mrapi_info_t version;

   // init the system
   mcapi_initialize(CNTRL_NODE, &err);
   CHECK_STATUS(err);

   mrapi_initialize(AUTO_USE_CASE_DOMAIN_ID, CNTRL_NODE,
                    parms,&version, &mrapi_status);
   CHECK_STATUS(mrapi_status);

   // first create local endpoints
   sig_endpt = mcapi_create_endpoint(CNTRL_PORT_SIG,
                                     &err);
   CHECK_STATUS(err);

   // now we get two rmem endpoints
   mcapi_get_endpoint_i(TPU_NODE, TPU_PORT_CNTRL,
                        &tpu_rmem_endpt, &r1, &err);
   CHECK_STATUS(err);

   mcapi_get_endpoint(SIG_NODE, SIG_PORT_CNTRL,
                      &sig_rmem_endpt, &r2, &err);
   CHECK_STATUS(err);

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

   // create our mutex for the shared memory region
   sMem_mutex =
        mrapi_mutex_create(SMEM_MUTEX_ID, MRAPI_NULL,
                           &mrapi_status);
   CHECK_STATUS(mrapi_status);

   // allocate shmem and send the handle to TPU task
   sMem = mrapi_shmem_create(SHMEM_ID, SHMEM_SIZE,
                             MRAPI_NULL, 0, MRAPI_NULL,
                             0, &mrapi_status);
   CHECK_STATUS(mrapi_status);

   sPtr = (void*) mrapi_shmem_attach(sMem,&mrapi_status);
   CHECK_STATUS(mrapi_status);
```

```
    tmp_endpt = mcapi_create_anonymous_endpoint(&err);
    CHECK_STATUS(err);

    // send the shmem handle
    mcapi_msg_send(tmp_endpt, tpu_rmem_endpt, sMem,
                   sizeof(sMem), &err);
    CHECK_STATUS(err);

    // connect the channels
    mcapi_connect_pktchan_i(sig_endpt, sig_rmem_endpt,
                            &r1, &err);
    CHECK_STATUS(err);

    // wait on the connection
    while (!mcapi_test(&r1,NULL,&err)) {
        // KEEP WAITING
    }

    // now open the channels
    mcapi_open_pktchan_recv_i(&sig_chan, sig_endpt,
                              &r1, &err);
    CHECK_STATUS(err);

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

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

    while (1) {
       // NOTE – get an MRAPI lock
       mrapi_mutex_lock (sMem_mutex, &lock_key, 0,
                         &mrapi_status);
       CHECK_STATUS(mrapi_status);

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

      // NOTE – release the MRAPI lock
      mrapi_mutex_unlock(sMem_mutex, &lock_key,
                         &mrapi_status);
      CHECK_STATUS(mrapi_status);

      // 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
    }
}
```

```
////////////////////////////////////////
// The TPU task
////////////////////////////////////////
void TPU_Task() {
   mrapi_shmem_hndl_t sMem;   /* handle to shmem */
   mrapi_mutex_hndl_t sMem_mutex;
   char* sPtr;
   mrapi_key_t lock_key;
   size_t msgSize;
   mcapi_endpoint_t cntrl_endpt;
   mcapi_request_t r1;
   mcapi_status_t err;

   // init the system
   mcapi_initialize(TPU_NODE, &err);
   CHECK_STATUS(err);

   mrapi_initialize(AUTO_USE_CASE_DOMAIN_ID, TPU_NODE,
                    MRAPI_NULL, MRAPI_NULL,&mrapi_status);
   CHECK_STATUS(mrapi_status);

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

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

   sPtr = (void*) mrapi_shmem_attach(sMem, &mrapi_status);
   CHECK_STATUS(mrapi_status);

   // ALL bootstrapping is finished, begin processing
   while (1) {

     // NOTE – get an MRAPI lock
     mrapi_mutex_lock(sMem_mutex, &lock_key, 0,
                      &mrapi_status);
     CHECK_STATUS(mrapi_status);

     // do something that updates shared mem
     sPtr[0] = 1;

    // NOTE – release the MRAPI lock
    void mrapi_mutex_unlock(sMem_mutex, &lock_key,
                            &mrapi_status);
   }
}
```

```
////////////////////////////////////////
// The SIG Processing Task
////////////////////////////////////////
void SIG_task() {
   mcapi_endpoint_t cntrl_endpt, cntrl_rmem_endpt;
   mcapi_pktchan_send_hndl_t cntrl_chan;
   mcapi_request_t r1;
   mcapi_status_t err;

   // init the system
   mcapi_initialize(SIG_NODE, &err);
   CHECK_STATUS(err);

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

   mcapi_get_endpoint_i(CNTRL_NODE, CNTRL_PORT_SIG,
                  &cntrl_rmem_endpt, &r1, &err);
   CHECK_STATUS(err);

   // wait on the rmem endpoint
   mcapi_wait(&r1,NULL,&err);
   CHECK_STATUS(err);

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

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


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

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

6.2.8.2 Changes required to port to new multicore device

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

6.3 Remote memory use cases

Use case 1 – accelerator core accesses host core's memory randomly using DMA and/or software cache

This use case aims to capture a common programming pattern for the Cell Broadband Engine processor. On the Cell processor the PPE (host) may launch a thread on an SPE (accelerator). The SPE can access (all of) PPE memory via DMA.

For certain kinds of access, particularly access to contiguous arrays in main memory, it is convenient to use DMA directly, using double or triple buffering to overlap communication with computation. This requires the use of *non-blocking* read and write operations on host memory.

For less regular types of access, but where there is likely to be some locality, it is common to use a software cache, which fetches data from main memory via DMA, but caches chunks of data locally to avoid repeated fetches. With a software cache, read and write calls are blocking.

The following use case illustrates these scenarios: we have two nodes, Node 1 and Node 2. Node 1 has a linked list data structure in memory, which Node 2 is going to process. For each element in the list, Node 2 will compute a score. The scores will be written back via DMA, as they are to be stored contiguously in Node 1's memory. The list elements will be read via a software cache as, we assume, they are relatively close together (e.g. perhaps the linked list elements are stored as an array, but are chained together in a non-contiguous order).

```
/*-----------------------------------------------*/
/* Definitions common to Node 1 and Node 2       */
/*-----------------------------------------------*/

typedef struct Entity_s {
      // DATA FIELDS (not specified here)
      struct Entity_s * next;
} Entity;

#define AGREED_ID_FOR_SW_CACHE 0 /* In a real application these IDs would
more likely be obtained */
#define AGREED_ID_FOR_DMA 1      /* by one node via MRAPI, and communicated
to the other node.    */




/*-----------------------------------------------*/
/* Node 1 side of use case                       */
/*-----------------------------------------------*/
```

```
/* Helper functions for Node 1 - these are not part of MRAPI, and could be
   implemented using various appropriate mechanisms */

/* Function which uses some mechanism (e.g. MCAPI) to send a message to Node
2 */
void send_to_node2(int);

/* Function via which Node 1 waits for Node 2 to complete its work */
void wait_for_notification_from_node2();

void* START_OF_HEAP;
int SIZE_OF_HEAP;

/* Code for Node 1 functionality */
int node1_remote_memory_use_case_1(Entity * entities_to_be_processed,
            float * scores_to_be_computed, unsigned int number_of_entities)

      /*
       'entities_to_be_processed' is a linked list of 'Entity' structures,
which are going to be processed by Node 2.
       Since the 'Entity' data is not contiguous, but elements of the list
may reside close together in memory, software
       caching is an appropriate access mechanism for the remote access.

       'scores_to_be_computed' is an array which is to be filled, by Node 2,
with a score for each entity.  Since
       elements of this array are contiguous, DMA is an appropriate access
mechanism for the remote access.

       */

{

      mrapi_status_t status; /* For error checking */

      /* Handles for software cache- and DMA-accessed remote memory */
      mrapi_rmem_hndl_t sw_cache_hndl;
      mrapi_rmem_hndl_t dma_hndl;

      /* We want Node 2 to process the linked list
'entities_to_be_processed'.  But elements of this list can be
         located anywhere on the heap, thus we need to make the heap
available remotely.
       */
      sw_cache_hndl = mrapi_rmem_create(AGREED_ID_FOR_SW_CACHE,
                                        START_OF_HEAP,
                                        MRAPI_ACCESS_TYPE_SW_CACHE,
                                        NULL,
                                        SIZE_OF_HEAP,
                                        &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
```

```
        ERR("Unable to create remote memory for sw cache");
      }
      /* Send Node 2, as integers, values of the pointers for
'entities_to_be_processed' and 'START_OF_HEAP' */
      send_to_node2((int)entities_to_be_processed);
      send_to_node2((int)START_OF_HEAP);

      /* Promote 'scores_to_be_computed' to allow remote access via DMA */
      dma_hndl = mrapi_rmem_create(AGREED_ID_FOR_DMA,
                                   scores_to_be_computed,
                                   MRAPI_ACCESS_TYPE_DMA,
                                   NULL,
                                   number_of_entities*sizeof(float),
                                   &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
          ERR("Unable to create remote memory for DMA");
      }
      /* Node 2 can now find these remote memory buffers, and work with them
*/

      /* Node 1 waits until Node 2 has finished (using some appropriate
mechanism) */
      wait_for_notification_from_node2();

      mrapi_rmem_detach(sw_cache_hndl, &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to detach from remote memory sw cache");
      }

      mrapi_rmem_delete(sw_cache_hndl, &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to delete remote memory for sw cache");
      }

      mrapi_rmem_detach(dma_hndl, &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to detach from remoty memory DMA");
      }

      mrapi_rmem_delete(dma_hndl, &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to delete remote memory for DMA");
```

```
      }

      return 0;

};




/*-------------------------------------------------*/
/* Node 2 side of use case                         */
/*-------------------------------------------------*/


/* Helper functions for Node 2 - these are not part of MRAPI, and could be
   implemented using various appropriate mechanisms */

/* Function to receive an integer message from Node 1, via some appropriate
mechanism (e.g. MCAPI) */
int receive_from_node1();

/* Function to determine whether an mrapi_rmem_get call succeeded */
int get_successful(mrapi_status_t*);

/* Function to tell Node 1 that processing has completed */
void notify_node1();

/* Function for doing processing on an 'Entity' */
float process(Entity *);


#define BUFFER_SIZE 1024

/* Buffers local to Node 2 used to store results, for double-buffered write-
back */
float result_buffers[2][BUFFER_SIZE];

int node2_remote_memory_use_case_1()
{
      mrapi_status_t status; /* For error checking */

      /* Handles for software cache- and DMA-accessed remote memory */
      mrapi_rmem_hndl_t sw_cache_hndl;
      mrapi_rmem_hndl_t dma_hndl;

      unsigned int start_of_node1_heap;
      unsigned int address_of_next_entity_to_process;

      start_of_node1_heap = receive_from_node1();
      address_of_next_entity_to_process = receive_from_node1();
```

```
      do {
            /* Get a handle to remote memory for software cache-access */
            sw_cache_hndl =
mrapi_rmem_get(AGREED_ID_FOR_SW_CACHE,MRAPI_ACCESS_TYPE_SW_CACHE,
                                                                  &status);
            // CHECK STATUS FOR ERROR
            if (status != MRAPI_SUCCESS) {
                  ERR("Unable to get remote memory for sw cache");
            }

      } while (!get_successful(&status));

      /* Use the handle to attach to the remote memory */
      mrapi_rmem_attach(sw_cache_hndl,
                        MRAPI_ACCESS_TYPE_SW_CACHE,
                        &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to attach to remote memory for sw cache");
      }

      do {
            /* Get a handle to remote memory for DMA-access */
            dma_hndl =
mrapi_rmem_get(AGREED_ID_FOR_DMA,MRAPI_ACCESS_TYPE_DMA,&status);

            // CHECK STATUS FOR ERROR
            if (status != MRAPI_SUCCESS) {
                  ERR("Unable to get remote memory for DMA");
            }

      } while (!get_successful(&status));

      /* Use the handle to attach to the remote memory */
      mrapi_rmem_attach(dma_hndl,MRAPI_ACCESS_TYPE_DMA,&status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
        ERR("Unable to attach to remote memory for DMA");
      }

      unsigned int num_entities_processed = 0;

      // Pair of request objects to allow us to wait for DMA operations on
      // either of two result buffers
      mrapi_request_t[2] request;

      Entity next_entity_to_process;

      bool first = true;
```

```
int cur_buf = 0; // Selects which buffer we are currently using.

do {
      unsigned int offset_for_next_entity =
       address_of_next_entity_to_process - start_of_node1_heap;

      /* Read an entity from Node 1's memory, via software cache.
      We use a blocking operation because we need the result to
      continue processing, and we hope the cache will mean that the
      result is held locally and will thus arrive quickly. */
      mrapi_rmem_read(sw_cache_hndl,
                              offset_for_next_entity,
                              &next_entity_to_process,
                              0,
                              sizeof(Entity),
                              1, /* num_strides is 1 */
                              0, /* rmem_stride is irrelevant */
                              0, /* local_stride is irrelevant */
                              &status);

      // CHECK STATUS FOR ERROR
      if (status != MRAPI_SUCCESS) {
            ERR("Unable to read remote memory sw cache");
      }

      result_buffers[cur_buf][num_entities_processed % BUFFER_SIZE] =
            process( & next_entity_to_process );
      num_entities_processed++;

      address_of_next_entity_to_process =
            (unsigned int)(next_entity_to_process.next);

      if((num_entities_processed % BUFFER_SIZE) == 0)
      {

            // CHECK STATUS FOR ERROR - DETAILS OMITTED

            /* Issue non-blocking DMA of buffer-full of results back
            to Node 1's memory.  We use a non-blocking operation
            because we do not need to wait for the write to
            complete in order to continue processing the list: it is
            preferrable to overlap communication with computation. */
            mrapi_rmem_write_i(
                        dma_hndl,
                        num_entities_processed*sizeof(float),
                        result_buffers[cur_buf],
                        0,
                        BUFFER_SIZE*sizeof(float),
                        1, /* num_strides is 1 */
                        0, /* rmem_stride is irrelevant */
                        0, /* local_stride is irrelevant */
```

```
                        &request[cur_buf],
                        &status);

                // CHECK STATUS FOR ERROR
                if (status != MRAPI_SUCCESS) {
                  ERR("Unable to initiate a remote memory write for DMA");
                }
                // Switch to use other buffer for processing, while
                // existing results are written back.
                cur_buf = 1 - cur_buf;

                /* Wait for previous write operation to complete */
                if(!first)
                {
                      mrapi_wait(&request[cur_buf], &status, NO_TIMEOUT);
                      if (status != MRAPI_SUCCESS) {
                        ERR("Unable to complete remote memory write DMA");
                      }
                } else {
                      first = false;
                }

        }

} while(next_entity_to_process.next != NULL);

/* Check to see if there is a partial buffer of results still to write
 back */
if((num_entities_processed % BUFFER_SIZE) != 0)
{

        // CHECK STATUS FOR ERROR - DETAILS OMITTED

        /* Issue non-blocking DMA of final results back to Node 1's
               memory */
        mrapi_rmem_write_i(dma_hndl,
              num_entities_processed*sizeof(float),
              result_buffers[cur_buf],
              0,
              (num_entities_processed % BUFFER_SIZE)*sizeof(float),
              1, /* num_strides is 1 */
              0, /* rmem_stride is irrelevant */
              0, /* local_stride is irrelevant */
              &request[cur_buf],
              &status);

        // CHECK STATUS FOR ERROR - DETAILS OMITTED
        if (status != MRAPI_SUCCESS) {
          ERR("Unable to initiate a remote memory write for DMA");
        }
        cur_buf = 1 - cur_buf;
```

```
        /* Wait for previous write operation to complete */
        if(!first)
        {
              mrapi_wait(&request[cur_buf], &status, NO_TIMEOUT);
              if (status != MRAPI_SUCCESS) {
                ERR("Unable to complete remote memory write for DMA");
              }
        }

  }

  /* Wait for final write operation to complete */
  mrapi_wait(&request[1-cur_buf], &status, NO_TIMEOUT);

  // CHECK STATUS FOR ERROR
  if (status != MRAPI_SUCCESS) {
        ERR("Unable to complete final remote memory write for DMA");
  }
  /* Detach from remote memories */

  mrapi_rmem_detach(sw_cache_hndl, &status);

  // CHECK STATUS FOR ERROR - DETAILS OMITTED
  if (status != MRAPI_SUCCESS) {
        ERR("Unable to detach from remote memory sw cache");
  }

  mrapi_rmem_detach(dma_hndl, &status);

  // CHECK STATUS FOR ERROR - DETAILS OMITTED
  if (status != MRAPI_SUCCESS) {
        ERR("Unable to detach from remote memory DMA");
  }

  /* Notify Node 1 that we are done */
  notify_node1();

  return 0;
  }
```

Remote Memory Use case 2

This use case captures the scenario where one processing node has a very small amount of RAM, and requires RAM on another processing node to be made available to store intermediate results. This is

common on a processor like Cell, where SPEs have just 256K local store, but the PPE is connected to a large main memory.

The idea in this use case is that Node 1 has a thread which is monitoring a thread on Node 2. The Node 1 thread sleeps until it receives a message from Node 2, either saying “stop”, or saying “I need more memory”. In the latter case, Node 1 allocates a new buffer of memory locally, and uses MRAPI to promote this to be remotely accessible, sending Node 2 the corresponding remote memory id. Node 2 can then use the memory as desired.

Once Node 2 completes, Node 1 can reclaim all the memory.

```
/*-----------------------------------------------*/
/* Definitions common to Node 1 and Node 2       */
/*-----------------------------------------------*/

/* Integer constants */
#define QUIT ...
#define MORE_DATA_PLEASE ...
#define AGREED_ACCESS_TYPE ...

/*-----------------------------------------------*/
/* Node 1 side of use case                       */
/*-----------------------------------------------*/

/* Helper functions for Node 1 - these are not part of MRAPI, and could be
   implemented using various appropriate mechanisms */

/* Function which uses some mechanism (e.g. MCAPI) to send a remote memory
id to Node 2 */
void send_id_to_node2(mrapi_rmem_id_t);

/* Function via which Node 1 waits for Node 2 to send an integer message */
void wait_for_message_from_node_2(int*);

/* Function via which Node 1 gets a fresh remote memory id */
mrapi_rmem_id_t get_fresh_rmem_id()


typedef struct Memory_Region_s
{
      char* pointer; /* A local buffer */
      mrapi_rmem_hndl_t handle; /* The remote memory handle associated with
this buffer */
} Memory_Region;

/* Array to keep track of memory which has been made available remotely */
Memory_Region buffers[MAX];


/* Code for Node 1 functionality */
int node1_remote_memory_use_case_2()
```

```
	/*
	Node 1 is "looking after" Node 2, and waits for Node 2 to send
	messages requesting more data.  On receiving such a message, Node 1
	allocates some more memory which it makes available to Node 2.
	Once Node 2 signals that it has completed, Node 1 deletes all the
	allocated memory.
	 */

{

	mrapi_status_t status; /* For error checking */

	int message;

	int next_buf = 0;

	while (wait_for_message_from_node_2(&message))
	{
		if(message == QUIT)
		{
			/* Node 2 says "I'm done", so we can exit the loop */
			break;
		}

		assert(message == MORE_MEMORY_PLEASE);

		/* Node 2 needs some more memory, and will have sent another
message saying how much */

		int amount_of_data_required_in_bytes;

		wait_for_message(&amount_of_data_required_in_bytes);

		/* Allocate the desired amount of memory locally */
		buffers[next_buf].pointer = (char*) malloc(
		 amount_of_data_required_in_bytes * sizeof(char) );

		/* We want to make this memory available remotely, so obtain an
id for the new piece of remote memory */
		mrapi_rmem_id_t id = get_fresh_rmem_id();

		/* Now promote the freshly allocated buffer to be visible
remotely */
		buffers[next_buf].handle = mrapi_rmem_create(
			id,
			buffers[next_buf].pointer,
			AGREED_ACCESS_TYPE,
			NULL,
			amount_of_data_required_in_bytes * sizeof(char),
			&status);

		// CHECK status FOR ERRORS - OMITTED
```

```
            /* Finally, tell Node 2 what the id is for the new memory */
            send_to_node2(id);

            next_buf++;

      }

      /* Node 2 has finished, so Node 1 can demote the memory regions it
made available remotely,
            and then free the corresponding memory
      */

      for(int i=0; i<next_buf; i++)
      {
            /* Demote piece of remote memory to no longer be remotely
visible */
            mrapi_rmem_delete(buffers[i].handle, &status);

            // CHECK status FOR ERRORS - OMITTED

            /* Now actually free the local memory which corresponded to this
remote memory */
            free(buffers[i].pointer);

      }

      return 0;

};




/*-------------------------------------------------*/
/* Node 2 side of use case                         */
/*-------------------------------------------------*/


/* Helper functions for Node 2 - these are not part of MRAPI, and could be
   implemented using various appropriate mechanisms */

/* Function which uses some mechanism (e.g. MCAPI) to receive a remote
memory id from Node 1 */
mrapi_rmem_id_t receive_id_from_node1();

/* Function which uses some mechanism (e.g. MCAPI) to send an integer
message to Node 1 */
void send_message_to_node_1(int);

int node2_remote_memory_use_case_2()
{
```

```
	mrapi_status_t status; /* For error checking */

	/* An array of remote memory handles */
	mrapi_hndl_t handles[MAX];

	int next_hndl = 0;

	while(...)
	{
		/* Node 2 does some processing which we do not specify here.
			Once in a while, Node 2 needs to use Node 1's memory as a
			"spill" buffer, thus requiring access to a region of this
			memory */

		...

		if( need to spill )
		{
			int number_of_bytes_required = ...;
			send_message_to_node_1(MORE_MEMORY_PLEASE);
			send_message_to_node_1(number_of_bytes_required);

			/* Node 1 will receive these messages and create some
				remotely accessible memory, for which it will send
an id */
			mrapi_rmem_id_t id = receive_id_from_node1();

			/* Use the id to get a handle for the remote memory */
			handles[next_hndl] = mrapi_rmem_get(id,
AGREED_ACCESS_TYPE,&status);

			// ERROR CHECKING ON status NOT SHOWN

			mrapi_rmem_attach(handles[next_hndl,
				AGREED_ACCESS_TYPE,
				&status);

			// ERROR CHECKING ON status NOT SHOWN

			next_hndl++;

			/* Now Node 2 can do some work using this remote memory,
via
				MRPI calls such as mrapi_rmem_read and
mrapi_rmem_write.
				We do not show details as this would be application-
specific
			*/
			...

		}
```

```
	}

	for(int i=0; i<next_hndl; i++)
	{
		mrapi_rmem_detach(handles[i], &status);

		// ERROR CHECKING ON status NOT SHOWN

	}

	send_message_to_node_1(QUIT);

	return 0;

}
```

6.4 Synchronization Use Case

TI has several chips that have a General Purpose Processor (GPP) and a DSP. The GPP traditionally runs a higher level RTOS like Linux, QNX, WinCE, etc. The DSP traditionally runs a DSP/BIOS.

One typical use case is to use the DSP as a video/audio accelerator. The GPP sends rmem processor call (RCP) messages to the DSP. An RCP message contains a pointer to the data to process, type and size of the data, how to process, etc. Once it is finished, the DSP sends a message back to complete the GPP RCP call.

The typical size of an RCP message is 4K bytes. The throughput is ~100 messages per second both ways (30 frames/second for video and 30-50 frames/second for audio). A typical system has ~64 messages in a system.

Generally the communication between the processors is either shared memory and interrupts or specialized hardware mechanisms. In the case of shared memory, the chips must support the same type of synchronization mechanism. The typical mechanisms include: spinlocks, hardware semaphores, support for Peterson's exclusion algorithm, and a few more.

Typically the GPP is the master and controls the starting/stopping of the DSP. All communication mechanisms must support the stopping of one side. The communication and synchronization mechanisms must also be portable to allow the easy migration of code to a different processor and OS.

6.5 Networking use case

Packet Processing Usage Case Patrick Griffin, Tilera

This use case extends the packet processing use case from the MCAPI specification to take advantage of MRAPI functions.

MRAPI's support for shared memory and mutexes is used to create a shared memory 'flow state' table, which tracks information about groups of packets flowing between the same source and destination hosts.

Use Case Description:

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 entrypoint 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 stream 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.

The worker's packet processing algorithm uses MRAPI to accomplish the following tasks:

1. At the start of time, it allocates a shared memory hash table that contains linked lists of packet flows.
2. As each packet arrives, the worker computes a hash bucket number based on its source and destination addresses.
3. The worker locks that hash bucket.
4. Having gained exclusive access to that bucket, the worker scans a linked list to see if the flow already exists, and if so, updates it.
5. If the flow is new to the system, the worker allocates a flow info object from MRAPI shared memory and adds it to the list.
6. The worker releases the lock on the hash bucket.

For ease of reference, the relevant MRAPI worker code is shown below. This code is run on several 'worker' cores, all accessing the same shared memory flow table in parallel. The code below includes both the shared memory initialization code run on all workers and the function that each worker calls when a packet arrives.

```
#define MY_SHMEM_ID 7

#define MAX_FLOWS (1024 * 1024)

#define HASH_BUCKETS 512


typedef struct flow_info_s {

  flow_info_s *next;

  ... various protocol state ...

} flow_info_t;


typedef struct {

  mrapi_mutex_hndl_t lock;

  flow_info_t *list;

} flow_hash_table_entry_t;
```

```
typedef struct {
  flow_hash_table_entry_t buckets[HASH_BUCKETS];
  flow_info_t* free_list_head;
  flow_info_t* free_list_tail;
  mrapi_mutex_hndl_t free_list_lock;
} flow_hash_table_t;

flow_hash_table_t *flow_table;



void init(int rank)
{
  mrapi_status_t status;

  if (rank == 0)
  {
    size_t size =
      sizeof(flow_info_t) * MAX_FLOWS +
      sizeof(flow_hash_table_t);

    mrapi_shmem_hndl_t mem_hndl =
      mrapi_shmem_create(MY_SHMEM_ID, size, NULL, 0, NULL, 0, &status);
    if (status != MRAPI_SUCCESS)
      die("Couldn't create shared memory.\n");

    void* mem = mrapi_shmem_attach(mem_hndl, &status);
    if (status != MRAPI_SUCCESS)
      die("Couldn't map shared memory region.\n");
```

```
  // This function takes our large allocation and fills in the hash
  // table.  In particular, it takes the large memory region and
  // splits into a flow_hash_table_t object and a free list
  // containing MAX_FLOWS flow_info_t objects.
  flow_table = init_flow_table(mem, MAX_FLOWS);

  // Initialize locks for each bucket, and for the free list.
  for (int i = 0; i < HASH_BUCKETS; i++)
  {
    flow_table->buckets[i] = mrapi_mutex_create(i, NULL, &status);
    if (status != MRAPI_SUCCESS)
      die("Couldn't allocate mutex for bucket.\n");
  }
  flow_table->free_list_lock =
    mrapi_mutex_create(HASH_BUCKETS + 1, NULL, &status);
  if (status != MRAPI_SUCCESS)
    die("Couldn't allocate mutex for free list.\n");
}

// This could be implemented using MCAPI.
barrier();

if (rank != 0)
{
  mrapi_shmem_hndl_t mem_hndl = mrapi_shmem_get(MY_SHMEM_ID, &status);
  if (status != MRAPI_SUCCESS)
    die("mrapi_shmem_get() failed.\n");

  flow_table = (flow_hash_table_t*) mrapi_shmem_attach(mem_hndl, &status);
  if (status != MRAPI_SUCCESS)
```

```
      die("Couldn't map shared memory region.\n");
  }
}

void update_flows(PacketInfo *packet) {
  int bucket = hash_packet(packet);
  flow_info_t* flow;

  mrapi_status_t status;
  mrapi_key_t lock_key;
  mrapi_mutex_lock(flow_table[bucket].lock, &lock_key, 0, &status);
  if (status != MRAPI_SUCCESS)
    die("Lock failure.\n");

  flow = scan_linked_list(flow_table[bucket].list, packet);
  if (!flow) {
    // Lock the free list and allocate a new flow.
    mrapi_key_t free_lock_key;
    mrapi_mutex_lock(flow_table->free_list_lock, &free_lock_key, 0,
&status);
    if (status != MRAPI_SUCCESS)
      die("Lock failure.\n");

    flow = alloc_from_free_list(flow_table);
    init_flow(flow, packet);
    add_to_linked_list(flow_table[table].list, flow);

    mrapi_mutex_unlock(flow_table->free_list_lock, &free_lock_key, &status);
    if (status != MRAPI_SUCCESS)
      die("Lock release failure.\n");
```

```
  }
  else
  {
    update_flow(flow, packet);
  }

  mrapi_mutex_unlock(flow_table[bucket].lock, &lock_key, &status);
    if (status != MRAPI_SUCCESS)
      die("Lock release failure.\n");
}
```

Again, the key MRAPI primitives are dynamic shared memory allocation and a mutex primitive. Statically allocated shared memory objects are used in the example code, but are not strictly required.

6.6 Metadata use cases

6.6.1 dynamic attribute example

Below is an example of monitoring a resource (L3 cache hits) and registering a callback event for when the counter rolls over.

```
mca_status_t mrapi_status;

#define WRONG wrong(__LINE__);
void wrong(unsigned line) {
  fprintf(stderr,"WRONG: line=%u status=%s\n",
line,mrapi_display_status(mrapi_status));
  fflush(stdout);
  exit(1);
}

/* Callbacks for handling when the counters rollover */
mrapi_boolean_t rollover = MRAPI_FALSE;
void l3cache_hits_rollover(void) {
  rollover = MRAPI_TRUE;
}

int main () {
  mrapi_parameters_t parms;
  mrapi_info_t version;
  mrapi_resource_t     *root;
  mrapi_rsrc_filter_t   filter;
  mrapi_resource_t     *l3cache;

  /* initialize */
  mrapi_initialize(DOMAIN,NODE,parms,&version,&mrapi_status);
```

```
  if (mrapi_status  != MRAPI_SUCCESS) { WRONG }

  /* Get the cache attributes */
  filter = MRAPI_RSRC_CACHE;
  root = mrapi_resources_get(filter, &mrapi_status);
  l3cache = root->children[0];
  uint32_t cache_hits;
  mrapi_resource_get_attribute(l3cache, 1, (void *)&cache_hits,
sizeof(cache_hits), &mrapi_status);
  if (mrapi_status != MRAPI_ERR_RSRC_NOTSTARTED) { WRONG }


  /* Start the L3 cache hit monitoring */
  mrapi_dynamic_attribute_start(l3cache, 1, &l3cache_hits_rollover,
&mrapi_status);
  if (mrapi_status != MRAPI_SUCCESS) { WRONG }

  while (rollover == MRAPI_FALSE) {
    mrapi_resource_get_attribute(l3cache, 1, (void *)&cache_hits, attr_size,
&mrapi_status);
    if (mrapi_status != MRAPI_SUCCESS) { WRONG }
    printf ("cache hits = %d",cache_hits);
  }

  /* stop the L3 cache hit monitoring */
  mrapi_dynamic_attribute_stop(l3cache, 1, &mrapi_status);
  if (mrapi_status != MRAPI_SUCCESS) { WRONG }
  mrapi_resource_get_attribute(l3cache, 1, (void *)&cache_hits, attr_size,
&mrapi_status);
  if (mrapi_status != MRAPI_ERR_RSRC_NOTSTARTED) { WRONG }

  /* finalize */
  mrapi_finalize(&mrapi_status);
  if (mrapi_status != MRAPI_SUCCESS) { WRONG }
```

6.6.2 mrapi_resource_get() examples

Below are a series of metadata use cases based on a single system. The use case figures are graphical representations of the resource data structure returned by a call to `mrapi_resources_get()`.

Consider as an example two CPUs and two memories connected by two busses, with a node running on each CPU.

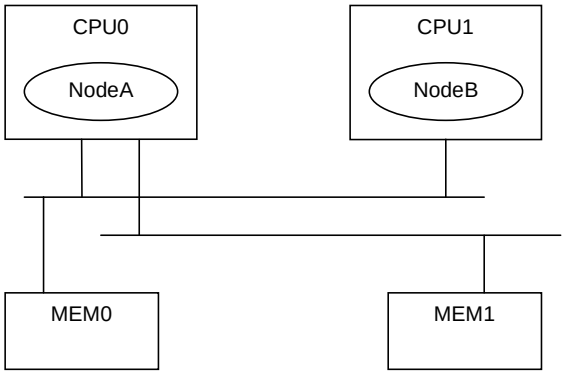

Figure 3 – Metadata Example Hardware

In Figure 3, CPU0 can access MEM0 and MEM1, and CPU1 can only access MEM0.

In the examples below, a hypervisor, which can partition the system, is included. The hypervisor can partition the system such that nodes running under guest operating systems can see only those units in the same partition.

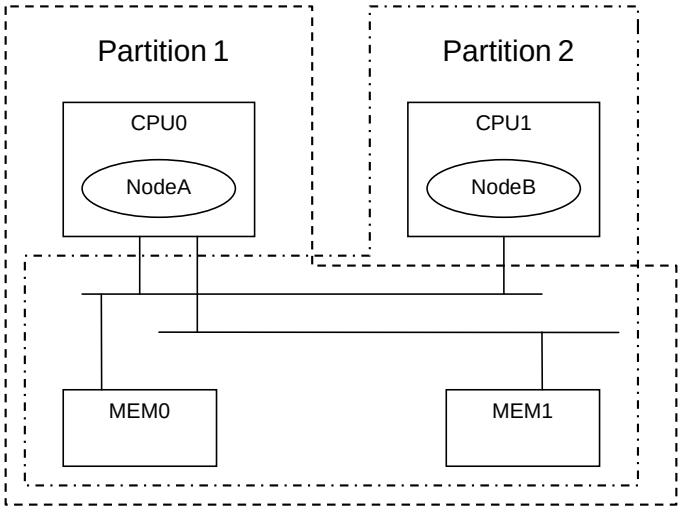

Figure 4 – Hypervisor Partitions of Example Hardware

Figure 4 shows a system with two partitions. Partition 1 has CPU0, MEM0, and MEM1 included. Partition 2 includes CPU1, MEM0, and MEM1. Since CPU0 and CPU1 are in different partitions, they are not visible to each other.

For the following situations, a graphical representation is given of the resource data structure.

1) Figure 5 shows a system with no partitions, with node A executing on CPU0, and “all” specified as the subsystem:

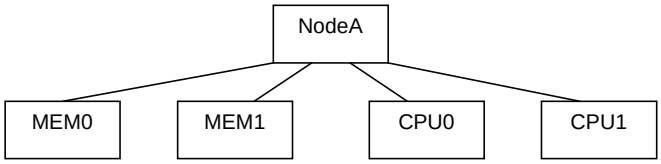

Figure 5 – Data Returned for Node A (Unpartitioned System)

2) Figure 6 shows a system with no partitions, node B executing on CPU1, and “all” specified as the subsystem:

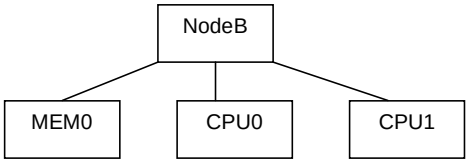

Figure 6 – Data for Node B (Unpartitioned System)

3) Figure 7 shows a system with partitions described above, node A executing on CPU0, and “all” specified as the subsystem:

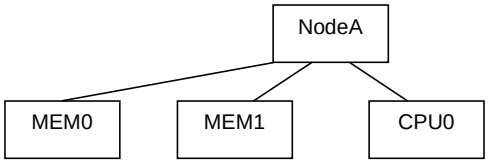

Figure 7 – Data Returned for Node A (Partitiioned System)

4) Figure 8 shows a system with the partitions described above, node B executing on CPU1, and “none” specified as the subsystem:

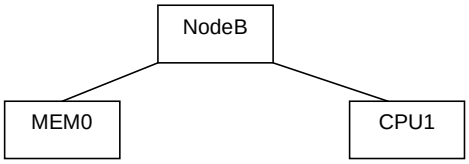

Figure 8 – Data Returned for Node B (Partitioned System)

5) Figure 9 shows Node A executing on CPU0, and the subsystem specified as “execution_context”, in order to determine which resource node A is executing on:

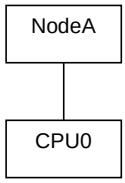

Figure 9 – Data Returned for Node A (Execution Context)

6) Figure 10 shows Node A executing on CPU0, no partition, with the subsystem specified as “memory”:

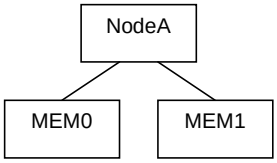

Figure 10 – Data Returned for Node A (Memory)

7) It is possible to register a callback function that is called when a system partition is changed. Suppose the system before re-partitioning is as in case (1) above, and then after re-partitioning CORE0 and CORE1 are on different partitions not visible to each other. A call to `mrapi_resources_get()` following the callback function (after re-partitioning) might yield the following. Using node A on CPU0, after re-partitioning, the results would look like case (3) above. Using node B on CPU1, after re-partitioning, the results would look like case (4) above.

Acknowledgements

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

Working Group
Sven Brehmer
Tasneem Brutch
Alastair Donaldson
Patrick Griffin
Jim Holt
Arun Joseph
Murat Karaorman
David Lindberg
Todd Mullanix
Stephen Olsen
Michele Reese
Andrew Richards
Ravi Singh
Roel Wuyts

The MRAPI 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
Daniel Forsgren
Masaki Gondo
Ganesh Gopalakrishnan
Marcus Hjortsberg
Paul Kelly
Tammy Lieno
Kenn Luecke
Anton Lokhmotov
Eric Mercer
Jarko Nissula
Ron Olson
Sabri Pllana
Cissy Yuan