Multicore Task Management API (MTAPI®) Specification V1.0

Document ID: MTAPI API Specification
Document Version: 1.0
Status: Release
Distribution: General

Copyright © 2013 The Multicore Association, Inc.

All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from The Multicore Association, Inc.

All copyright, confidential information, patents, design rights and all other intellectual property rights of whatsoever nature contained herein are and shall remain the sole and exclusive property of Multicore Association. The information furnished herein is believed to be accurate and reliable. However, no responsibility is assumed by The Multicore Association, Inc. for its use, or for any infringements of patents or other rights of third parties resulting from its use.

The Multicore Association, Inc. name and The Multicore Association, Inc. logo are trademarks or registered trademarks of The Multicore Association, Inc. All other trademarks are the property of their respective owners.

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

Table of Contents

Preface

This document is intended to assist software developers who are either implementing MTAPI or writing applications that use MTAPI.

MTAPI was developed under the guidance of The Multicore Association (MCA) with participation by many of the MCA member companies. This MTAPI specification fits within the roadmap defined by the MCA. The foundation components of that roadmap are the Multicore Communications API (MCAPI) and the Multicore Resource API (MRAPI). MTAPI, MCAPI, and MRAPI share many concepts, constructs, and goals.

Definitions

Action	An *action* is the hardware or software implementation of a *job*. An action implemented in software consists of the implementation of an *action function* with a predefined signature. Software actions are registered with the MTAPI runtime and associated with a job. While executing, an action is also associated with a task and task context. Hardware implementations of actions must be known a priori in the MTAPI runtime implementation. There is no standardized way of registering hardware actions because hardware implementations are highly hardware dependent. Hardware and software actions are referenced by handles or indirectly through job IDs and job handles.
Action Function	The callable, executable function of an action, invoked by the MTAPI runtime when a task is started.
Affinity	Defines which cores can execute a given action function.
AMP	Asymmetric multiprocessing, in which two or more processing cores having the same or different architecture may be running the same or different operating systems (or no OS at all).[1]
API	Application Programming Interface.
Blocking	A blocking function does not return until the function completes successfully or returns with an error.
Core	A core is an undividable processing element. Two cores can share resources such as memory, or ALUs for hyperthreaded cores. The core notion is necessary for core affinity, but is implementation-specific.
Data Parallelism	Data Parallelism is a programming model focusing on distributing data to different computing nodes (processors, processor cores, or parallel vector processing units). Since many languages and programming libraries express data parallelism as parallel loops, data parallelism often appears as *loop-level parallelism*.

[1] Not to be confused with “C++ AMP”, a library provided by Microsoft. In this case, “AMP” is an acronym for "Accelerated Massive Parallelism."

Domain	An implementation of MTAPI includes one or more domains, each with one or more nodes. The concept of domains is consistent in all Multicore Association APIs. A domain is comparable to a subnet in a network or a namespace for unique names and IDs. Domains are supported by a runtime.
Handle	An abstract reference to an object on the same node or to an object managed by another node. A handle is valid only on the node on which it was requested and generated. A handle is opaque, that is, its underlying representation is implementation-defined. Handles can be copied, assigned, and passed as arguments, but the application should make no other assumptions about the type, representation, or contents of a handle.
ISA	Instruction Set Architecture.
Job	A job provides a way to reference one or more actions. Jobs are abstractions of the processing implemented in hardware or software by actions. Multiple actions can implement the same job based on different hardware resources (for instance a job can be implemented by one action on a DSP and by another action on a general purpose core, or a job can be implemented by both hardware and software actions). Each job is represented by a domain-wide job ID, or by a job handle local to a node.
MCA	The Multicore Association.
MCAPI	*Multicore Communications API Specification*, defined by The Multicore Association.
MRAPI	*Multicore Resource Management API Specification*, defined by The Multicore Association.
MTAPI	*Multicore Task Management API Specification*, defined by The Multicore Association.
Node	A node represents an independent unit of execution that maps to a process, thread, thread pool, instance of an operating system, hardware accelerator, processor core, a cluster of processor cores, or other abstract processing entity with an independent program counter. Each node can belong to only one domain. The concept of nodes is consistent in all Multicore Associations APIs. Code executed on an MTAPI node shares memory (data) with any other code executed on the same node.
OS	Operating System.
POSIX	Portable Operating System Interface, an API for software compatibility with Unix, specified by the IEEE.
Queue	A software or hardware entity in which tasks are enqueued in a given order. The queue can ensure in-order execution of tasks. Furthermore, queues might implement other scheduling policies that can be configured by setting queue attributes.
Reference	In this document, a reference exists when an object or abstract entity has knowledge or access to another object, without regard to the specific means of implementation.
Resource	A processing core or chip, hardware accelerator, memory region, or I/O.
Remote Memory	Memory that cannot be accessed using standard load and store operations. For example, host memory is remote to a GPU core.

Runtime System	An MTAPI runtime system (or “runtime”) is the underlying implementation of MTAPI. The core of the runtime system supports task scheduling and communication with other nodes. Each MTAPI has an MTAPI runtime system.
SMP	SMP is short for symmetric multiprocessing, in which two or more identical processing cores are connected to a shared main memory and are controlled by a single OS instance.
SoC	System-on-chip.
Task	A task is the invocation of an action. A task is associated with a job object, which is associated with one or more actions. A task may optionally be associated with a task group. A task has attributes and an internal state. A task begins its lifetime with a call to `mtapi_task_start()` or `mtapi_task_enqueue()`. A task is reference by a handle of type `mtapi_task_hndl_t`. After a task has started, it is possible to wait for task completion from other parts of the program. Every task can run exactly once, i.e., the task cannot be started a second time. (Note that outside of this document, the term “task” has a different meaning. Some real-time operating systems use “task” for operating system threads, for example.)
Task Context	Information about the task, accessible by the corresponding action function; useful for action code reflection.
Task Parallelism	In contrast to data parallelism, task parallelism focuses on distributing work (tasks) to parallel computing nodes.

Related Documents

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

1. Introduction

1.1 Overview

This Multicore Task API (MTAPI) specification defines an API for application-level management of tasks in multicore embedded systems. Task parallel programming allows a programmer to split a program into tasks that are executed in parallel. In contrast to threads, tasks are significantly lighter weight, allowing systems to execute hundreds or thousands of parallel tasks.

The major difference between MTAPI and other task parallel APIs is its design for homogeneous as well as heterogeneous systems (i.e., MTAPI supports distributed memory architectures, multicore systems, and multi-processor systems with processor cores implementing different instruction set architectures). The second unique feature of MTAPI is its design for very resource-constrained devices. The MTAPI specification is designed for minimal implementations in plain C which can be run on top of an embedded operating system or even in a bare metal environment.

MTAPI is scalable and can support virtually any number of cores, each with a different processing architecture and each running the same or a different operating system, or no OS at all. As such, MTAPI is intended to provide source code compatibility that allows applications to be ported from one operating environment to another well into the future.

1.1.1 MTAPI Goals

The Multicore Association continues to tear down barriers that impede development of complex multicore applications. Using homogeneous or heterogeneous multicore processors requires the programmer to develop software that splits a software program into tasks that can be executed in parallel on different processor cores. Today’s operating systems and runtime libraries for embedded systems provide threads or thread-like mechanisms that are not suited for the fine-grain parallelism required by multicore architectures, typically because the coordination of hundreds or thousands of parallel threads generates too much overhead relative to the actual computation time. The current programming model requires complex, low-level synchronization and programming with threads and is limited to single operating systems running on single homogeneous multicore processors. In heterogeneous embedded systems, however, system-wide task management is needed.

MTAPI aims to eliminate these obstacles by providing an API that allows programmers to develop parallel embedded software in a straightforward manner. Core features of MTAPI are runtime scheduling and mapping of tasks to processor cores. Due to its dynamic behavior, MTAPI is intended for optimizing throughput on multicore systems. Furthermore, it provides the means to allow the software developer to adjust the task scheduling strategy for specific requirements (for example, to focus more on latency or fairness).

Unlike existing APIs that provide task management functionality (e.g., OpenMP [4][9], TBB [5], Cilk [2]), MTAPI allows implementations for resource-constrained embedded systems, such as those with a small memory footprint, deterministic behavior, and hardware-specific optimizations. Furthermore, portability is essential for the implementation. Therefore, MTAPI supports different processor architectures and can be implemented on top of different operating systems or as a bare-metal solution. In short, MTAPI supports asymmetric multiprocessing at the hardware and software level. MTAPI (in conjunction with other Multicore Association APIs) can serve as a valuable tool for implementing applications. For these reasons, we developed the MTAPI features to meet the following goals

- Small application-layer API, suitable for cores on a chip and chips on a board.
- Easy to learn and use.
- Incorporates an essential feature set.

- Supports lightweight and high-performance implementations.
- Does not prevent use of complementary approaches.
- Allows silicon providers to optimize their hardware.
- Allows implementers to differentiate their offerings.
- Can run on top of an OS, hypervisor, or bare metal.
- Can utilize hardware acceleration.
- Supports hardware implementations of the API.
- Does not require homogeneous cores, operating system, or memory architecture.
- Supports source-code portability.

MTAPI provides easy to use programming models that can be used directly by the application programmer for embedded systems with limited CPU and memory resources. Furthermore MTAPI provides a common portability layer that helps with implementing higher-level programming APIs on top, such as OpenMP and TBB.

1.1.2 The MTAPI Feature Set

MTAPI supports two programming modes derived from use cases of the working group members (Figure 1):

- **Tasks**
  MTAPI allows a programmer to start tasks and to synchronize on task completion. Tasks are executed by the runtime system, concurrently to other tasks that have been started and have not been completed at that point in time. A task can be implemented by software or by hardware. Tasks can be started from remote nodes, i.e., the implementation can be done on one node, but the starting and synchronization of corresponding tasks can be done on other nodes. The developer decides where to deploy a task implementation. On the executing node, the runtime system selects the cores that execute a particular task. This mapping can be influenced by application-specific attributes. Tasks can start sub-tasks. MTAPI provides a basic mechanism to pass data to the node that executes a task, and back to the calling node.

- **Queues**
  Explicit queues can be used to control the task scheduling policies for related tasks. Order-preserving queues ensure that tasks are executed sequentially in queue order with no subsequent task starting until the previous one is complete. MTAPI also supports non-order-preserving queues, allowing control of the scheduling policies of tasks started via the same queue (queues may offer implementation specific scheduling policies controlled by implementation specific queue attributes). Even hardware queues can be associated with queue objects.

MTAPI also supports the following types of tasks:

- **Single tasks**
  Single tasks are the standard case: After a task is started, the application may wait for completion of the task at a later point in time. In some cases the application waits for completion of a group of tasks. In other cases waiting is not required at all. When a software-implemented task is started, the corresponding code (action function) is executed once by the MTAPI runtime environment. When a hardware-implemented task is started, the task execution is triggered once by the MTAPI runtime system.

- **Multi-instance tasks**
  Multi-instance tasks execute the same action multiple times in parallel (similar to parallel regions in OpenMP or parallel MPI processes).

- **Multiple-implementation tasks / load balancing**
  In heterogeneous systems there could be implementations of the same job for different types of processor cores, e.g., one general purpose implementation and a second one for a hardware accelerator. MTAPI allows attaching multiple actions to a job. The runtime system shall decide dynamically during runtime, depending on the system load, which action to utilize. Only one of the alternative actions will be executed.

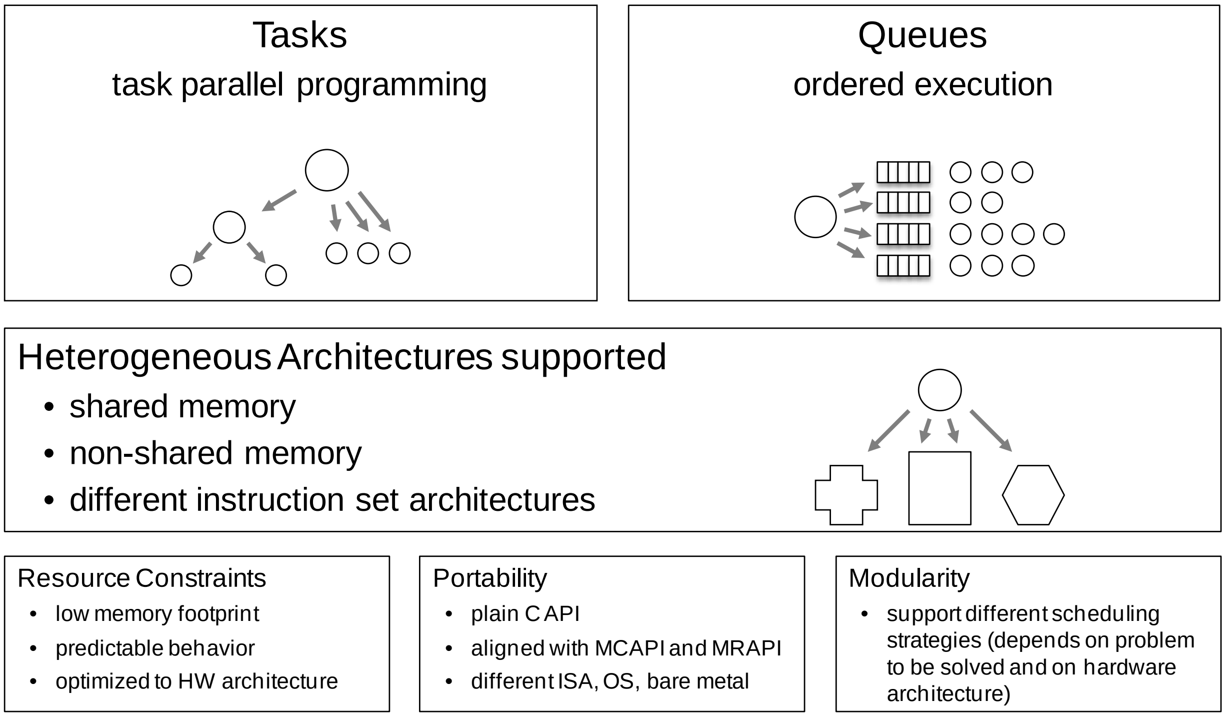
Tasks
task parallel programming
Queues
ordered execution
Heterogeneous Architectures supported
• shared memory
• non-shared memory
• different instruction set architectures
Resource Constraints
• low memory footprint
• predictable behavior
• optimized to HW architecture
Portability
• plain C API
• aligned with MCAPI and MRAPI
• different ISA, OS, bare metal
Modularity
• support different scheduling strategies (depends on problem to be solved and on hardware architecture)

Figure 1: MTAPI programming models

1.1.3 MTAPI Reference Architecture

Figure 2 shows an overview of the MTAPI architecture. The application uses the MTAPI interface and optionally also MRAPI and MCAPI interfaces. For some application domains, libraries providing high-level APIs are built on top of MTAPI. In this case the application might not use MTAPI directly.

The core of the MTAPI runtime system implementation is a scheduler that controls task execution. Optionally the implementation provides access to hardware-implemented actions and/or queues. MRAPI can be used as part of MTAPI's internal portability layer. The same applies for MCAPI, which can be used for inter-node communication internally.

The lower layers provide the mapping to hardware and operating system resources (e.g., threads). MTAPI can be implemented on bare metal, on top of an operating system, or directly on top of a hypervisor (which is not shown in Figure 2).

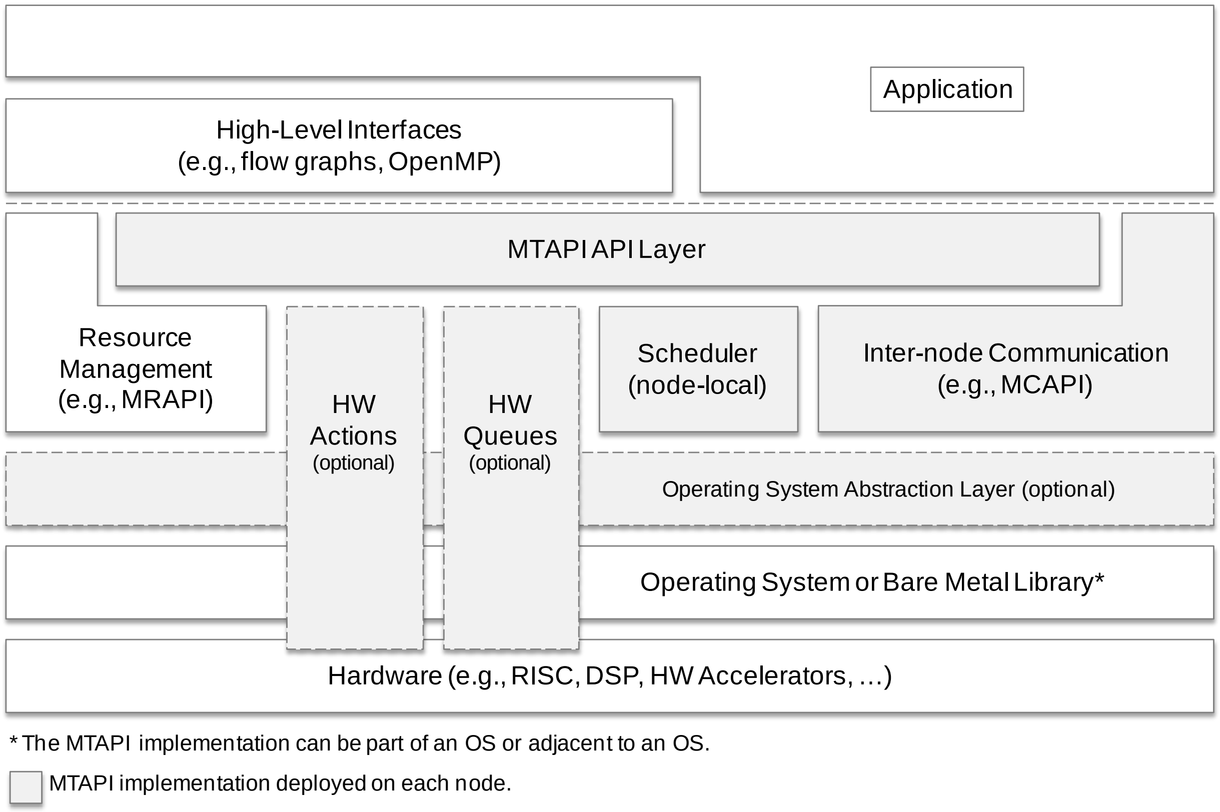
Application
High-Level Interfaces (e.g., flow graphs, OpenMP)
MTAPI API Layer
Resource Management (e.g., MRAPI)
HW Actions (optional)
HW Queues (optional)
Scheduler (node-local)
Inter-node Communication (e.g., MCAPI)
Operating System Abstraction Layer (optional)
Operating System or Bare Metal Library*
Hardware (e.g., RISC, DSP, HW Accelerators, …)
* The MTAPI implementation can be part of an OS or adjacent to an OS.
MTAPI implementation deployed on each node.

Figure 2: MTAPI Reference Architecture

Figure 3 shows the communication of two MTAPI runtime system instances on different nodes of a system. The communication needed for task management is hidden by the MTAPI implementation. It is transparent to the application if an action is implemented on the node where the corresponding task is started, or if it is executed on a remote node.

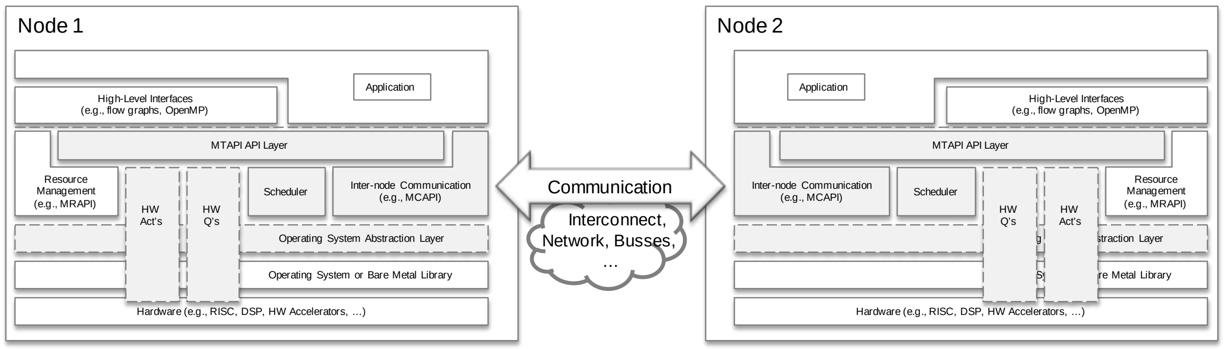
Node 1
Application
High-Level Interfaces (e.g., flow graphs, OpenMP)
MTAPI API Layer
Resource Management (e.g., MRAPI)
HW Act's
HW Q's
Scheduler
Inter-node Communication (e.g., MCAPI)
Operating System Abstraction Layer
Operating System or Bare Metal Library
Hardware (e.g., RISC, DSP, HW Accelerators, …)
Communication
Interconnect, Network, Busses, …
Node 2

Figure 3: MTAPI inter-node communication

1.2 History and Related APIs

Multicore programming shares many concepts with parallel and distributed computing, whereby 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 operating systems enabled for Symmetric Multiprocessing (SMP).

As multicore computing extends into embedded domains, many aspects of computing heterogeneity emerge. This limits the ability of programmers to use 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 task 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 significant additional requirements not encompassed by existing standards.

MTAPI 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.

The MTAPI programming model was heavily influenced by existing standards and libraries addressing concurrent and parallel programming, resulting in an API that can be used comfortably by an application programmer. Unlike other standards and libraries, MTAPI concentrates on core functionality in order to keep implementations small. This enables MTAPI implementations to be used for very resource-limited devices.

MTAPI allows the implementation of existing programming interfaces, such as OpenMP, on top of MTAPI. This would enhance the portability of these parallel programming models by using MTAPI as a porting layer closer to the hardware. The main aim of this porting layer is to abstract away most of the low-level details of specific hardware components and improve programmer productivity.

The MTAPI working group addressed specific areas of functionality related to the following existing standards.

1.2.1 POSIX Threads, Mutexes, and Semaphores

Shared memory provides multiple threads of execution access to the same data, which may be on the same processor or on multiple processors, thereby avoiding copying of the data. The Portable Operating System Interface (POSIX)[2] standard 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 more than one operating system process, and one or more processor cores.

Shared memory and synchronization, covered by MRAPI, is considered essential for multicore programming. The MTAPI working group added support for the following:

[2] http://standards.ieee.org/develop/wg/POSIX.html

- Heterogeneous (AMP) systems
- Lightweight tasks in order to run thousands of tasks in parallel
- Easy synchronization between tasks and groups of tasks
- Influencing the order of execution (e.g., for stream processing)

The POSIX standard provides threads and two forms of semaphores: mutexes (binary semaphores), and semaphores (counting semaphores). Given the goals of MRAPI, the MRAPI working group considered POSIX mutexes and semaphores (IEEE Standard 1003.1b) as having relevant functionality. It was decided to have MRAPI cover a subset of POSIX mutex and semaphore functionality along with the following new requirement (see MRAPI specification). The MRAPI working group determined that condition variables and signaling should be considered within the scope of the MTAPI working group rather than the MRAPI working group. Condition variables and signaling were not considered in this version of MTAPI due to the more abstract approach of parallel tasks in the sense of *task-parallel programming,* rather than the approach as *multi-threading* in the sense of threads or task implemented by an operating system. A second reason for not including condition variables and signaling was the lack of use cases needing such features in a heterogeneous environment, as, in a shared memory SMP environment, condition variables and signaling is provided by the operation system, Pthreads, or by the standard C11 library. Therefore, the functionality should be considered on the Multicore Association roadmap, but deferred until adequate use cases are available.

1.2.2 C11

C11 is an informal name for ISO/IEC 9899:2011[3] [7] that introduces the following features (among others) for shared memory programming:

- Threads
- Condition variables
- Mutexes
- Thread-specific storage

Regarding MTAPI, the same statements as mentioned for Pthreads and POSIX semaphores can be applied.

1.2.3 OpenMP

OpenMP[4] (Open Multi-Processing) is a portable application programming interface (API), which supports shared-memory parallel programming in C/C++ and Fortran [4][9]. It consists of a set of compiler directives, library routines, and environment variables that control runtime behavior. OpenMP has been implemented in many commercial compilers from organizations such as Intel, Sun, PGI, IBM, and Cray, as well as in open source compilers such as GCC and OpenUH from the University of Houston. OpenMP is designed to permit incremental parallelization of existing code, in contrast to other programming paradigms like message passing, and allows both the sequential and the parallel version of a program in a single source file in order to simplify maintenance. Jointly defined by a group of major computer hardware and software vendors, OpenMP is available on many platforms, including Windows and Linux.

OpenMP offers a task-based programming model. Developers express logical parallelism in a given application using tasks with the runtime library adopting several effective strategies to schedule these tasks to the worker threads. This strategy is in close association with the feature set being discussed for MTAPI. Currently (2012/2013) discussions are taking place on proposed extensions for accelerator support.

---

[3] http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=57853

[4] http://www.openmp.org

The advantage of using MTAPI instead of OpenMP is the independence from the compiler. Built as a programming library, MTAPI can be easily compiled for any embedded system having a C compiler. Furthermore MTAPI provides support for light-weight task based programming in heterogeneous systems.

The portability of these parallel programming models like OpenMP can be enhanced by using MTAPI as a porting layer as described at the beginning of section 1.1.3.

1.2.4 Intel® Threading Building Blocks

Intel® Threading Building Blocks (Intel® TBB)[5] is a C++ template library designed to simplify the development of parallel programs that take advantage of multicore processors [5][6][10]. It exploits data- as well as task-level parallelism and aims to relieve the developer from the complexity of traditional thread-based approaches. For that purpose, Intel TBB provides data structures and algorithms that hide the complexity of thread management and synchronization. In this way, data races and other complications arising in multi-threaded applications can often be avoided. As another advantage, Intel TBB abstracts from platform-dependent details such as the underlying threading package (e.g., Windows threads, POSIX threads, etc.) and the number of processor cores. As a result, applications written for the architectures of today are likely to scale with an increasing number of cores in the architectures of tomorrow.

Due to the STL style, the code utilizing Intel TBB is more complex compared to OpenMP or Cilk (see [11], for example). This affects programming effectiveness and leads to a higher risk for programming errors.

Intel TBB has a high awareness level in shared memory systems and some of the concepts of Intel TBB are similar to MTAPI concepts. Intel TBB, beyond that, provides some higher level programming models, (e.g., flow graphs), that simplify application development in certain domains and for certain problems; however it not designed for low-footprint and heterogeneous systems. Similar programming models can be realized on top of MTAPI in a portable way.

1.2.5 Parallel Patterns Library

Microsoft's Parallel Patterns Library[6] (PPL), released in 2010, provides a task scheduler, parallel loops, and data structures optimized for concurrent access by several threads, with core features which are similar to Intel TBB, PPL is proprietary and limited to Microsoft’s operating systems. The library is not designed for embedded systems, nor is it portable. Beside the fact that PPL is not designed for embedded systems and the source code is unavailable, the same statements as for Intel TBB apply. MTAPI can be used as a foundation for portable implementation of higher level APIs.

1.2.6 Grand Central Dispatch

Grand Central Dispatch (GCD)[7] is an API introduced by Apple and available on iOS and OS X operating systems. It basically provides a thread pool to take advantage of task-level parallelism provided by symmetric multiprocessing systems. The GCD programming model is based on different types of queues: sequential queues ensuring the order of task execution, and global queues for concurrent execution. MTAPI provides a similar programming model using sequential and concurrent queues. In contrast with GCD, MTAPI is specifically designed for heterogeneous embedded systems.

[5] http://software.intel.com/en-us/intel-tbb

[6] http://msdn.microsoft.com/en-us/library/dd492418.aspx

[7] https://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html

1.2.7 Cilk, Cilk++, Intel® Cilk™ Plus

Cilk[8] [12] is an algorithmic multithreaded language. The philosophy behind Cilk is that a programmer should concentrate on structuring the program to expose parallelism and exploit locality, leaving Cilk's runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Thus, the Cilk runtime system takes care of details like load balancing, paging, and communication protocols. Unlike other multi threaded languages, however, Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. It is based on ANSI C and was created at MIT. The following snippet shows a short code example:

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```

The simplicity of Cilk was an inspiration for the core parts of MTAPI, however MTAPI covers heterogeneous systems while Cilk is restricted to shared memory environments.

Cilk Arts[9] was a spin-off commercializing and advancing Cilk to Cilk++. It differs from Cilk in several ways: support for C++, operation with both Microsoft and GCC compilers, support for loops, and "Cilk hyperobjects" – a new construct designed to solve data race problems created by parallel accesses to global variables [13]. Cilk Arts was acquired by Intel in 2009 and the successor of Cilk++ was renamed to Intel® Cilk™ Plus.

1.2.8 OpenCL

OpenCL is a standard maintained by the Khronos Group[10] designed for data parallel processing. It is mainly used for general purpose computation on graphics processing units (GPGPU). In order to support data parallel processing on a wide range of accelerators, the OpenCL programming model is built on top of an abstract memory model which leads to a certain degree of complexity: host memory, global memory, constant memory, local memory, private memory.

MTAPI, in contrast, focuses on task parallelism with a much smaller and streamlined API that is easily portable and extensible for any kind of embedded architecture. The node abstraction provides the possibility to also have different kinds of processors and memory regions in the system. In addition OpenCL requires an OpenCL compiler while MTAPI, as mentioned, is implemented as a compiler independent library.

[8] http://supertech.csail.mit.edu/cilk/

[9] http://www.cilk.com/

[10] http://www.khronos.org/

2. MTAPI Concepts

The major MTAPI concepts are covered in the following sections. The concepts and supporting data types are defined to meet the goals stated in Section 1.1.1, including source code portability.

2.1 MTAPI Design Goals and Constraints

Complexity and Implementation Effort

MTAPI is designed to be a low-level API providing core functionality. The complexity is kept at minimum in order to keep the hurdle low for providing MTAPI implementations.

MCA API Consistency

MTAPI syntax and semantics are aligned to the MCAPI and MRAPI specifications, i.e., the core concepts of domains and nodes are part of MTAPI as well. Furthermore, the philosophy of creating objects and setting attributes was adopted from the aforementioned specifications.

Portability of Applications

Applications and libraries implemented using MTAPI shall be highly portable to systems with different hardware architecture (i.e., different number of processors or cores, different types of cores, such as accelerators).

Performance

The MTAPI design allows high-performance implementations with simple memory management. MTAPI supports hardware-implemented queues, actions, and multi-instance scheduling. Furthermore, MTAPI provides extension points (e.g., implementation-specific task and queue attributes) that allow utilizing hardware-specific features. Of course, these features lower portability and therefore should be used in systems only when portability decrease is acceptable because of performance gain.

Porting Effort of Existing Applications

It should be easy to port existing applications to MTAPI. In particular, porting Pthreads-based code should be straightforward.

2.2 Domain

The concept of a domain is shared amongst all Multicore Association APIs, and it must be consistent (1) within any implementation that supports multiple APIs, and (2) across implementations that require interoperability. MTAPI supports domains for compatibility with the MCA API specifications MCAPI and MRAPI.

An MTAPI system is composed of one or more MTAPI *domains*. An MTAPI domain is a unique system global entity. Each MTAPI domain comprises a set of MTAPI *nodes* (Section 2.3). An MTAPI node may only belong to one MTAPI domain, while an MTAPI domain may contain one or more MTAPI nodes. This allows the programmer to use MTAPI domains as namespaces for all kinds of IDs (e.g., nodes, actions, queues, etc.).

2.3 Nodes

An MTAPI node is an independent unit of execution, such as a process, thread, thread pool, processor, hardware accelerator, or instance of an operating system. A given MTAPI implementation specifies what constitutes a node for that implementation.

The intent is to avoid a mixture of node definitions in the same implementation (or in different domains within an implementation). If a node is defined as a unit of execution with its private address space (like a process), then a core with a single unprotected address space OS is equivalent to a node, whereas a core with a virtual memory OS can host multiple nodes.

On a shared memory SMP processor, a node can be defined as a subset of cores. A quad-core processor, for example, could be divided into two nodes, one node representing three cores and one node representing the fourth core reserved exclusively for certain tasks.

The definition of a node is flexible because this allows applications to be written in the most portable fashion supported by the underlying hardware, while at the same time supporting more general-purpose multicore and many-core devices. The definition allows portability of software 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 concept of nodes in MTAPI is shared with other Multicore Association API specifications. Therefore, implementations that support other 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.

2.4 Tasks and Actions

2.4.1 Tasks

A *task* represents the computation associated with the data to be processed. A task is executed concurrently to the code starting the task. The main API functions are `mtapi_task_start()` (Section 3.8.3) and `mtapi_task_wait()` (Section 3.8.7). The semantics are similar to the corresponding thread functions (e.g. `pthread_create`/`pthread_join` in Pthreads). The lifetime of a task is limited; it can be started only once.

2.4.2 Actions

In order to cope with heterogeneous systems and computations implemented in hardware, a task is not directly associated with an entry function as it is done in other task-parallel APIs. Instead, it is associated with at least one *action object* representing the calculation. The association is indirect: one or more actions implement a job, one job is associated with a task. If the action is implemented in software, this is either a function on the same node (which can represent the same processor or core) or a function implemented on a different node that does not share memory with the core starting the task (Figure 4).

Starting a task consists of three steps:

1. Create the action object with a job ID (software-implemented actions only).
2. Obtain an job reference.
3. Start the task using the job reference.

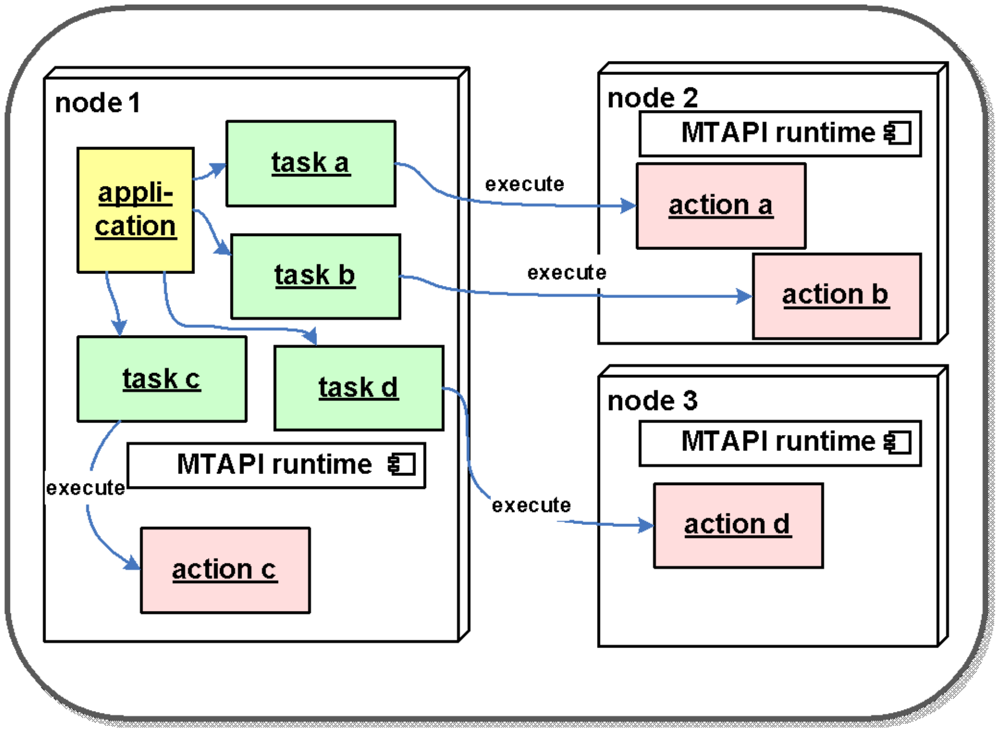

Figure 4: Tasks and Actions

2.4.3 Synchronization

The basic synchronization mechanism provided within MTAPI is waiting for task completion. Calling `mtapi_task_wait()` (Section 3.8.7) with a task handle blocks the current thread[11] or task until the task referenced by the handle has completed. Depending on the implementation, the calling thread can be used for executing other tasks while waiting for the task to be completed.

In order to synchronize with a set of tasks, every task can be associated with a *task group*. The MTAPI methods `mtapi_group_wait_all()` and `mtapi_group_wait_any()` (Sections 3.9.6 and 3.9.7) wait for a group of tasks or completion of any task in the group, respectively.

MTAPI only provides synchronization on task granularity. Synchronization inside a task implementation can be done by MCAPI messages, MRAPI synchronization primitives, and the MRAPI memory primitives. If MCAPI or MRAPI implementations are not available, synchronization mechanisms provided by the operating system or a threading library must be used. In this case, the MTAPI implementation must define the consequences of using those mechanisms in the task context.

2.5 Queues

Queues are used for guaranteeing sequential order of execution of tasks. A common use case is packet processing in the communication domain: for every connection all packets must be processed sequentially, while the packets of different connections can be processed in parallel to each other.

Sequential execution is accomplished by using a queue for every connection and queuing all packets of one connection into the same queue. In some systems, queues are implemented in hardware,

[11] Tasks can be started from an operating system thread.

otherwise MTAPI implements software queues. MTAPI is designed for handling thousands of queues that are processed in parallel.

The procedure for setting up and using a queue is as follows:

1. Create the action object (software-implemented actions only).

2. Obtain a job reference.

3. Create a queue object and attach the job to the queue (software-implemented queues only).

4. Obtain a queue handle if the queue was created on a different node, or if the queue is hardware-implemented.

5. Use the queue: enqueue the work using the queue.

Another important purpose of queues is that different queues can express different scheduling attributes for the same job. For example, in contrast to order-preserving queues, non-order-preserving queues can be used for load-balancing purposes between different computation nodes. In this case, the queue must be associated with more than one action implementing the same task on different nodes (i.e., different processors or cores implementing different instruction set architectures). If a queue is configured this way, the order will not be preserved.

2.6 Attributes

Attributes are provided as a means to extend the API. Different implementations may define and support additional attributes beyond those predefined by the API. To promote portability and implementation flexibility, attributes are maintained in an opaque data object that may not be directly examined by the user. Each object (e.g., task, action, queue) has an attributes data object associated with it, and many attributes have a small set of predefined values that must be supported by MTAPI implementations. The user may initialize, get, and set these attributes.

For default behavior, it is not necessary to call the initialize, get, and set attribute functions. However, to get non-default behavior, the typical four-step process is:

1. Declare an attributes object of the `mtapi_<object>_attributes_t` data type.
2. `mtapi_<object>attr_init()`: Returns an attributes object with all attributes set to their default values.
3. `mtapi_<object> attr _set()`: (Repeat for all attributes to be set). Assigns a value to the specified attribute of the specified attributes object.
4. `mtapi_<object>_create()`: Passes the attributes object modified in the previous step as a parameter when creating the object.

At any time, the user can call `mtapi_<object>_get_attribute()` to query the value of an attribute. After an object has been created, some objects allow to change attributes by calling `mtapi_<object>_set_attribute()`.

2.7 Error Handling Philosophy

Error handling is a fundamental part of the MTAPI specification. However, some accommodations have been made to support 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 values of several API functions indicate success or failure, which enables efficient coding practice. A parameter of type `mtapi_status_t` will encode success or failure states of API calls. `MTAPI_NULL` is a not valid return value for `mtapi_status_t`. If used in a C++ environment, exception must not leave an action function.

MTAPI provides timeouts for the `mtapi_task_wait()`, `mtapi_group_wait_all()` , and `mtapi_group_wait_any()` functions, and an `mtapi_task_cancel()` (Section 3.8.6) function to clear outstanding non-blocking requests at the non-failing side. It is also possible to reinitialize a failed node by first calling `mtapi_finalize()`(Section 3.2.5).

2.8 Timeout and Cancellation Philosophy

MTAPI provides timeout functionality for its non-blocking calls through the timeout capability of the `mtapi_task_wait()`, `mtapi_group_wait_all()` , and `mtapi_group_wait_any()` functions. Many blocking-function implementations have `timeout_t` parameters. Setting the timeout to `MTAPI_INFINITE` means a function call will not time out, setting it to a constant 0 or the symbolic constant `MTAPI_NOWAIT` causes the function to return at once (non-blocking call).

2.9 Data Types

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

In general, API parameters that refer to MTAPI 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. A handle is only valid on the node where it was created or where it was obtained. Passing handles between nodes results in undefined behavior. Create and get functions require MTAPI *ID* types (see Sections 2.9.2, 2.9.3, 2.9.12) to be passed in and will return a handle (see Sections 2.9.4, 2.9.7) for use in all other function calls related to that MTAPI object.

2.9.1 Scalars

MTAPI defines its own integer, Boolean and other types, some of which have MCA equivalents. See Appendix B: Header Files on page 130 of this document for specifics on these data types.

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

- `mtapi_uint64_t`
- `mtapi_uint32_t`
- `mtapi_uint16_t`
- `mtapi_uint8_t`
- `mtapi_int64_t`
- `mtapi_int32_t`
- `mtapi_int16_t`
- `mtapi_int8_t`

The following additional types define platform-specific integers depending on the implementation:

- `mtapi_int_t`
- `mtapi_uint_t`

2.9.2 General Types

mtapi_status_t

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

mtapi_timeout_t

The `mtapi_timeout_t` type is used to indicate the duration that an `mtapi_task_wait()`, `mtapi_group_wait_all()` , and `mtapi_group_wait_any()` API call will block before reporting a timeout. The units of the `mtapi_timeout_t` data type are implementation-defined because mechanisms for time keeping vary from system to system. Applications should not rely on this feature for satisfaction of realtime constraints because its use will not guarantee application portability across MTAPI implementations. The `mtapi_timeout_t` data type is intended only to allow for error detection and recovery. The implementation must define the reserved values `MTAPI_NOWAIT` to not block at all, and `MTAPI_INFINITE` for infinite waiting.

mtapi_size_t

The `mtapi_size_t` type is an unsigned integral type used to indicate the size of data. This could be the number of elements or the size of a data structure in bytes.

2.9.3 Domains and Nodes

mtapi_domain_t

The `mtapi_domain_t` type is used for MTAPI domains. The domain id scheme is implementation-defined. For application portability we recommend using symbolic constants. The `mtapi_domain_t` type has an `mca_domain_t` equivalent.

mtapi_node_t

The `mtapi_node_t` type is used for MTAPI nodes. The exact underlying type is implementation-defined. The node numbering is implementation-defined. For application portability we recommend using symbolic constants. The `mtapi_node_t` type has an `mca_node_t` equivalent.

mtapi_info_t

Initialization parameters allow implementations to configure the MTAPI runtime. A parameter allows implementations to provide information about the MTAPI runtime — both MTAPI-specified and implementation-specific information.

The informational parameters include MTAPI-specified information as outlined below, as well as implementation-specific information. The implementer must document implementation-specific information.

MTAPI-defined initialization information is stored in an object of type `mtapi_info_t`, which shall be implemented as a `struct` type. The `mtapi_info_t` struct shall define the following members, however, the specific underlying types of these members are implementation-defined:

- `mtapi_version`: MTAPI version. The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.
- `organization_id`: Implementation vendor or organization ID.
- `implementation_version`: Vendor version. The three last (rightmost) hex digits are the minor number, and those left of the minor number are the major number.
- `number_of_domains`: Number of domains allowed by the implementation.
- `number_of_nodes`: Number of nodes allowed by the implementation.

2.9.4 Handles

mtapi_job_hndl_t

The `mtapi_job_hndl_t` type is a reference to all actions implementing the same job. The implementations can be on the current node or on a remote node. The job handle is used to create tasks that execute an action implementation referenced by the handle.

mtapi_action_hndl_t

The `mtapi_action_hndl_t` type is a reference to the implementation of a job. The action handle is used to modify action attributes, to enable or disable actions, and to delete actions.

mtapi_task_hndl_t

The `mtapi_task_hndl_t` type is used for operations on tasks, i.e., waiting for tasks and cancelling tasks.

mtapi_queue_hndl_t

The `mtapi_queue_hndl_t` type is used to start tasks via a queue. This allows ensuring the order of task execution or to control common scheduling strategies for similar tasks.

mtapi_group_hndl_t

The `mtapi_group_hndl_t` type is used to wait for completion of a group of tasks.

2.9.5 Identifiers

mtapi_job_id_t

The `mtapi_job_id_t` type is used to get handles to the associated job obejcts. 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, `mtapi_job_id_t` there is a pair of corresponding identifiers in the MTAPI header file – `MTAPI_MIN_USER_JOB_ID` and `MTAPI_MAX_USER_JOB_ID` – that can be examined by the application writer to determine valid ID extents. Thus, user-specified IDs can range from `MTAPI_MIN_USER_JOB_ID` to `MTAPI_MAX_USER_JOB_ID`.

mtapi_queue_id_t

The `mtapi_queue_id_t` type is used to get handles to the associated queue obejcts. 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, `mtapi_queue_id_t` there is a pair of corresponding identifiers in the MTAPI header file – `MTAPI_MIN_USER_QUEUE_ID` and `MTAPI_MAX_USER_QUEUE_ID` – that can be examined by the application writer to determine valid ID extents. Thus, user-specified IDs can range from `MTAPI_MIN_USER_QUEUE_ID` to `MTAPI_MAX_USER_QUEUE_ID`.

mtapi_task_id_t

MTAPI types allow setting identifiers of tasks for debugging purposes (they may be set to `MTAPI_TASK_ID_NONE` if not used).

The implementation defines what is invalid. For any identifier, `mtapi_task_id_t` there is a pair of corresponding identifiers in the MTAPI header file – `MTAPI_MIN_USER_TASK_ID` and `MTAPI_MAX_USER_TASK_ID` – that can be examined by the application writer to determine valid ID extents. Thus, user-specified IDs can range from `MTAPI_MIN_USER_TASK_ID` to `MTAPI_MAX_USER_TASK_ID`.

mtapi_group_id_t

MTAPI types allow setting identifiers of task groups for debugging purposes (they may be set to `MTAPI_GROUP_ID_NONE` if not used).

The implementation defines what is invalid. For any identifier, `mtapi_group_id_t` there is a pair of corresponding identifiers in the MTAPI header file – `MTAPI_MIN_USER_GROUP_ID` and `MTAPI_MAX_USER_GROUP_ID` – that can be examined by the application writer to determine valid ID extents. Thus, user-specified IDs can range from `MTAPI_MIN_USER_GROUP_ID` to `MTAPI_MAX_USER_GROUP_ID`.

2.9.6 Action Functions and Action Function Context

mtapi_action_function_t

The `mtapi_action_function_t` represents a function pointer to an action function. The prototype is defined in chapter 3.4 on page 50.

mtapi_task_context_t

The `mtapi_task_context_t` type is used for obtaining information about the task in the action. For example, it is used to obtain the number of parallel tasks, and it is used to write the task results.

mtapi_affinity_t

The `mtapi_affinity_t` type is an opaque object representing a task-to-core affinity mask.

mtapi_task_state_t

The `mtapi_task_state_t` type represents a task state. A minimal implementation must support

- `MTAPI_TASK_CANCELLED`
- `MTAPI_TASK_RUNNING`

Other states may be

- `MTAPI_TASK_CREATED`
- `MTAPI_TASK_SCHEDULED`

- `MTAPI_TASK_WAITING`
- `MTAPI_TASK_DELETED`
- `MTAPI_TASK_COMPLETED`

mtapi_notification_t

The `mtapi_notification_t` type is an opaque type used for implementation-specific MTAPI extensions. This can be used for influencing scheduling behavior from the context of the action code. Possible extensions are:

- explicit prefecting of data for successor tasks
- relaxing strict ordered execution in queued processing after a critical part of the action has been completed

2.10 Sharing Across Nodes

Tasks and task groups can be used node-locally only. Queues and actions can be referenced globally by IDs. The IDs are used to obtain node-local handles. Handles in general cannot be copied between nodes.

2.11 Sharing Across Domains

The following MTAPI primitives are shared across domains by default: jobs and queues. Implementations may suffer a performance impact for resources that are shared across domains. For any of these primitives, you can disable sharing across domains by setting the `MTAPI_DOMAIN_SHARED` attribute to `MTAPI_FALSE` and passing it to the corresponding `*create()` function. If sharing is disabled for an object, handles for this object are not obtainable in a different domain.

2.12 Application Portability Concerns

The MTAPI standard was developed to enable application portability but cannot guarantee it. The reason is that a 100% application portability would massively restrict hardware-dependent optimization which shall still be possible using MTAPI. The guiding principles that should be used by application writers are:

- Write as much of the application in as portable a fashion as possible.
- Encapsulate 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 MTAPI 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. A given node should not know or care whether another node, with which it is interacting, is implemented in hardware or software.

The MTAPI 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.

2.13 Implementation Concerns

2.13.1 Task Mapping

Task execution on the node, where the corresponding action function is implemented should use all software and hardware features that lead to a good utilization of the cores belonging to the node. Implementations on top of an operating system, for example, should map task execution to operating systems threads. Systems that have to support hundreds and thousands of tasks should use appropriate thread pool implementations (e.g., work-stealing schedulers). The provider of the implementation has to specify the scalability of the implementation regarding the number of tasks and queues.

2.13.2 Restricted Feature Sets

One MTAPI implementation may consist of several runtime systems with reduced feature sets. This allows keeping implementations minimal. If a function is not supported on a particular runtime system, the status is set to either of the following:

- `MTAPI_ERR_FUNC_NOT_IMPLEMENTED`, if the MTAPI function called is not implemented by the runtime system, or

- `MTAPI_ERR_ARG_NOT_IMPLEMENTED`, if the MTAPI function called is implemented by the runtime, but it does not support the arguments passed.

Example 1: heterogeneous SoC

A SoC that consists of a general purpose core and a DSP core should implement the complete MTAPI API for the general purpose core. If the DSP is only used for executing tasks triggered by the general purpose processor, the DSP runtime can only support the execution of tasks triggered by the general purpose core, i.e., the DSP does not support `mtapi_task_start()` and related calls (start, enqeue, wait, task group functions).

Example 2: shared memory SMP

Pure shared memory systems might not support remote tasks. In many cases it is sufficient to always create actions on the node where they are started or enqueued. This also applies to queues in this case.

2.13.3 Memory Management

MTAPI requires minimal memory allocation on heap during runtime. In order to support dynamic highly task parallel applications there is runtime memory allocation for administrative data structures at least in three situations:

- Creation of queues (data objects representing queues: elements, status and attribute settings).

- Creation of task groups (data objects representing task groups).

- Start of tasks (data objects representing tasks: status and attribute settings).

The reason for this is in the dynamic nature of many task-parallel applications. In systems that do not allow dynamic memory allocation (i.e., memory allocation only during an initialization phase or static memory allocation), memory for these calls has to be pre-allocated. This can either be handled by providing a simple memory manager, allocating pre-allocated chunks of memory in the above-mentioned cases or by passing pointers to pre-allocated memory by implementation-specific attributes.

2.13.4 Thread-Safe Implementations

If an MTAPI implementation is available in a threaded environment, then the MTAPI implementation must be thread-safe. MTAPI 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.

2.14 Potential Future Extensions

With the goal of implementing MTAPI efficiently, the API has been kept simple. This has the potential for adding more functionality on top of MTAPI later. Some specific areas for adding functionality include higher level programming models such as flow graphs and data parallelism, and informational functions for debugging, statistics (optimization), and status. These areas are strong candidates for future extensions, and they are briefly described in the following subsections.

2.14.1 Condition Variables and Signaling

The MRAPI specification provides low-level synchronization mechanisms including mutexes and semaphores. MRAPI considered condition variables and signaling to be within the scope of MTAPI. The MTAPI working group decided to focus on a pure task-parallel programming model, supporting fine grain tasks and synchronizing on task completion. This allows highly efficient implementations mapping task execution to thread pool-based schedulers or hardware implementations. Therefore condition variables and signaling are not part of the current MTAPI specification, but they still are candidates for future extensions.

2.14.2 Flow Graphs

Many algorithms, if modeled as a graph of dependent functions, can take advantage of parallel execution (e.g., wavefront algorithms or FFT). Each node of the graph triggers a task that executes the associated function, possibly parallel to tasks triggered by other nodes. A flow graph is set up once during an initialization phase. During runtime, data items are processed consecutively by the graph.

Flow graphs define dependencies between functions. A vertex (or node) of a flow graph represents a function, and the edges define control flow dependencies. The intention is to execute all functions that do not have control flow dependencies in parallel. Intel Threading Building Blocks (Intel TBB) introduces a flow graph API in Intel TBB 4.0. In comparison, MTAPI provides only a basic support for simple flow graphs.

A simple flow graph API based on MTAPI would be based on the following dependency definitions:

- Start a task when a certain task completed.
- Start several tasks after a certain task completed.
- Start a task when a set of tasks completed.
- Start several tasks when a set of tasks completed.

2.14.3 Data Parallelism

Data-parallel operations are common in APIs used for data-intensive calculations like image processing. Those APIs either provide parallel for-loops or operations on arrays. Loop iterations or array elements are processed in parallel. MTAPI currently does not include data-parallel operations. They would be helpful especially for hardware-supported parallel processing (vector units, for example). However, as current implementations show, those operations highly depend on the underlying hardware (memory hierarchy, types of vectors supported, etc.). For software-based implementations, data-parallel operations can easily be built on top of MTAPI by breaking loop iterations into multiple independent tasks.

3. MTAPI API

The MTAPI API is divided into eight major parts:

- General
- Actions
- Action Functions and Action Context Functions
- Core Affinities
- Queues
- Jobs
- Tasks
- Task Groups

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

3.1 Conventions

`MTAPI_IN` and `MTAPI_OUT` are used to distinguish between input (parameter will be read) and output (parameter will be written) parameters. Parameters that are read and written are declared as `MTAPI_INOUT`.

3.2 General

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

3.2.1 MTAPI_NODEATTR_INIT

NAME

```
mtapi_nodeattr_init()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_nodeattr_init(
  MTAPI_OUT mtapi_node_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function initializes a node attributes object. A node attributes object is a container of node attributes. It is an optional argument passed to `mtapi_initialize()` to specify non-default node attributes when creating a node.

To set node attributes to non-default values, the application must allocate a node attributes object of type `mtapi_node_attributes_t` and initialize it with a call to `mtapi_nodeattr_init()`. The application may call `mtapi_nodeattr_set()` to specify attribute values. Calls to `mtapi_nodeattr_init()` have no effect on node attributes after the node has been created and initialized with `mtapi_initialize()`. The `mtapi_node_attributes_t` object may safely be deleted by the application after the call to `mtapi_nodeattr_init()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.

SEE ALSO

`mtapi_nodeattr_set()`
`mtapi_initialize()`

3.2.2 MTAPI_NODEATTR_SET

NAME

```
mtapi_nodeattr_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_nodeattr_set(
  MTAPI_INOUT mtapi_node_attributes_t* attributes,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function sets node attribute values in a node attributes object. A node attributes object is a container of node attributes, optionally passed to `mtapi_initialize()` to specify non-default node attributes when creating a node.

`attributes` is a pointer to a node attributes object that was previously initialized with a call to `mtapi_nodeattr_init()`. Calls to `mtapi_nodeattr_set()` have no effect on node attributes after the node has been created and initialized with `mtapi_initialize()`. The node attributes object may safely be deleted by the application after the call to `mtapi_initialize()`.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

MTAPI-defined node attributes:

Attribute num	Description	Data Type	Default
`MTAPI_NODES_NUMCORES`	(Read-only) number of processor cores of the node.	`mtapi_uint_t`	(no default)

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.

SEE ALSO

`mtapi_nodeattr_init()`
`mtapi_initialize()`

3.2.3 MTAPI_INITIALIZE

NAME

```
mtapi_initialize()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_initialize(
  MTAPI_IN mtapi_domain_t domain_id,
  MTAPI_IN mtapi_node_t node_id,
  MTAPI_IN mtapi_node_attributes_t* attributes,
  MTAPI_OUT mtapi_info_t* mtapi_info,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

`mtapi_initialize()` initializes the MTAPI environment on a given MTAPI node in a given MTAPI domain. It must be called on each node using MTAPI. A node maps to a process, thread, thread pool, instance of an operating system, hardware accelerator, processor core, a cluster of processor cores, or other abstract processing entity with an independent program counter. In other words, an MTAPI node is an independent thread of control.

Application software running on an MTAPI node must call `mtapi_initialize()` once per node. It is an error to call `mtapi_initialize()` multiple times from a given node, unless `mtapi_finalize()` is called in between.

The values for `domain_id` and `node_id` must be known a priori by the application and MTAPI.

`mtapi_info` is used to obtain information from the MTAPI implementation, including MTAPI and the underlying implementation version numbers, implementation vendor identification, the number of cores of a node, and vendor-specific implementation information. See the header files for additional information.

A given MTAPI implementation will specify what is a node, i.e., how the concrete system is partitioned into nodes and what are the underlying units of execution executing tasks, e.g., threads, a thread pool, processes, or hardware units.

`attributes` is a pointer to a node attributes object that was previously prepared with `mtapi_nodeattr_init()` and `mtapi_nodeattr_set()`. If `attributes` is `MTAPI_NULL`, then implementation-defined default attributes will be used.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_NODE_INITFAILED`	The MTAPI environment could not be initialized.
`MTAPI_ERR_NODE_INITIALIZED`	The MTAPI environment has already been initialized.
`MTAPI_ERR_NODE_INVALID`	The `node_id` parameter is not valid.
`MTAPI_ERR_DOMAIN_INVALID`	The `domain_id` parameter is not valid.
`MTAPI_ERR_PARAMETER`	Invalid `mtapi_node_attributes` or `mtapi_info` parameter.

SEE ALSO

`mtapi_finalize()`
`mtapi_nodeattr_init()`

3.2.4 MTAPI_NODE_GET_ATTRIBUTE

NAME

`mtapi_node_get_attribute()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_node_get_attribute(
  MTAPI_IN mtapi_node_t node,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Given a node and attribute number, returns a copy of the corresponding attribute value in `*attribute`. See `mtapi_nodeattr_set()` for a list of predefined attribute numbers and the sizes of the attribute values. The application is responsible for allocating sufficient space for the returned attribute value and for setting `attribute_size` to the exact size in bytes of the attribute value.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value will be written to `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined.

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_nodeattr_set()`

3.2.5 MTAPI_FINALIZE

NAME

```
mtapi_finalize()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_finalize(
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

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

All tasks that have not completed and that have been started on the node where `mtapi_finalize()` is called will be cancelled (see `mtapi_task_cancel()`). `mtapi_finalize()` blocks until all tasks that have been started on the same node return. (Long-running tasks already executing must actively poll the task state and return if cancelled.) Tasks that execute actions on the node where `mtapi_finalize()` is called, also block finalization of the MTAPI runtime system on that node. They are cancelled as well and return with a `MTAPI_ERR_NODE_NOTINIT` status. Other functions that have a dependency to the node and that are called after `mtapi_finalize()` also return `MTAPI_ERR_NODE_NOTINIT` (e.g., `mtapi_task_get()` starting a task associated with an action implemented on the already-finalized node).

`mtapi_finalize()` may not be called from an action function.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_NODE_FINALFAILED`	The MTAPI environment could not be finalized.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_initialize()`
`mtapi_task_cancel()`

3.2.6 MTAPI_DOMAIN_ID_GET

NAME

```
mtapi_domain_id_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_domain_t mtapi_domain_id_get(
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns the domain id associated with the local node.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

3.2.7 MTAPI_NODE_ID_GET

NAME

```
mtapi_node_id_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_node_t mtapi_node_id_get(
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

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

RETURN VALUE

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

ERRORS

`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

3.3 Actions

A *task* is a particular invocation of a *job*. A job refers to one or more *actions*. An action is a hardware or software implementation of a job. An action is referenced by an opaque handle of type `mtapi_action_hndl_t`, or indirectly through a handle to a job of type `mtapi_job_hndl_t`. A job refers to all actions implementing the same job, regardless of the node(s) where they are implemented.

An action's lifetime begins when the application successfully calls `mtapi_action_create()` and obtains a handle to the action. Its lifetime ends upon successful completion of `mtapi_action_delete()` or `mtapi_finalize()`.

While an opaque handle to an action may be used in the scope of one node only, a job can be used to refer to all its associated actions implementing the same job, regardless of the node where they are implemented. Tasks may be invoked in this way from nodes that do not share memory or even the same ISA with the node where the action resides.

3.3.1 MTAPI_ACTIONATTR_INIT

NAME

```
mtapi_actionattr_init()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_actionattr_init(
  MTAPI_OUT mtapi_action_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function initializes an action attributes object. A action attributes object is a container of action attributes, optionally passed to `mtapi_action_create()` to create an action with non-default attributes.

The application is responsible for allocating the `mtapi_action_attributes_t` object and initializing it with a call to `mtapi_actionattr_init()`. The application may then call `mtapi_actionattr_set()` to specify action attribute values. Calls to `mtapi_actionattr_init()` have no effect on action attributes after the action has been created with `mtapi_action_create()`. The `mtapi_action_attributes_t` object may safely be deleted by the application after the call to `mtapi_action_create()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_actionattr_set()`
`mtapi_action_create()`

3.3.2 MTAPI_ACTIONATTR_SET

NAME

```
mtapi_actionattr_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_actionattr_set(
  MTAPI_INOUT mtapi_action_attributes_t* attributes,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function sets action attribute values in an action attributes object. An action attributes object is a container of action attributes, optionally passed to `mtapi_action_create()` to create an action with non-default attributes.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

Calls to `mtapi_actionattr_set()` have no effect on action attributes after the action has been created. The `mtapi_action_attributes_t` object may safely be deleted by the application after the call to `mtapi_action_create()`.

MTAPI-defined action attributes:

Attribute num	Description	Data Type	Default
`MTAPI_ACTION_GLOBAL`	Indicates whether or not this is a globally visible action. Local actions are not shared with other nodes.	`mtapi_boolean_t`	`MTAPI_TRUE`
`MTAPI_ACTION_AFFINITY`	Core affinity of action code.	`mtapi_affinity_t`	`all cores set`
`MTAPI_DOMAIN_SHARED`	Indicates whether or not the action is shareable across domains.	`mtapi_boolean_t`	`MTAPI_TRUE`

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_actionattr_init()`
`mtapi_affinity_set()`
`mtapi_action_create()`

3.3.3 MTAPI_ACTION_CREATE

NAME

`mtapi_action_create()`

SYNOPSIS

```
#include <mtapi.h>

mtapi_action_hndl_t mtapi_action_create(
  MTAPI_IN mtapi_job_id_t job_id,
  MTAPI_IN mtapi_action_function_t function,
  MTAPI_IN void* node_local_data,
  MTAPI_IN mtapi_size_t node_local_data_size,
  MTAPI_IN mtapi_action_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function creates a software action (hardware actions are considered to be pre-existant and do not need to be created).It is called on the node where the action function is implemented. An action is an abstract encapsulation of everything needed to implement a job. An action contains attributes, a reference to a job, a reference to an action function, and a reference to node-local data. After an action is created, it is referenced by the application using a node-local handle of type `mtapi_action_hndl_t`, or indirectly through a node-local job handle of type `mtapi_job_hndl_t`. An action's lifecycle begins with `mtapi_action_create()`, and ends when `mtapi_action_delete()` or `mtapi_finalize()` is called.

To create an action, the application must supply the domain-wide job ID of the job associated with the action. Job IDs must be predefined in the application and runtime, of type `mtapi_job_id_t`, which is an implementation-defined type. The job ID is unique in the sense that it is unique for the job implemented by the action. However several actions may implement the same job for load balancing purposes.

For non-default behavior, `*attributes` must be prepared with `mtapi_actionattr_init()` and `mtapi_actionattr_set()` prior to calling `mtapi_action_create()`. If `attributes` is `MTAPI_NULL`, then default attributes will be used.

If `node_local_data_size` is not zero, `node_local_data` specifies the start of node local data shared by action functions executed on the same node. `node_local_data_size` can be used by the runtime for cache coherency operations.

RETURN VALUE

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

ERRORS

MTAPI_ERR_JOB_INVALID	The job_id is not a valid job ID, i.e., no action was created for that ID or the action has been deleted.
MTAPI_ERR_ACTION_EXISTS	This action is already created.
MTAPI_ERR_ACTION_LIMIT	Exceeded maximum number of actions allowed.
MTAPI_ERR_ACTION_NOAFFINITY	The action was created with an MTAPI_ACTION_AFFINITY attribute that has set the affinity to all cores of the node to MTAPI_FALSE.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_ERR_PARAMETER	Invalid attributes parameter.

SEE ALSO

mtapi_actionattr_init()
mtapi_actionattr_set()
mtapi_action_create()

3.3.4 MTAPI_ACTION_SET_ATTRIBUTE

NAME

```
mtapi_action_set_attribute()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_action_set_attribute(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function changes the value of the attribute that corresponds to the given `attribute_num` for this action.

`attribute` must point to the attribute value, and `attribute_size` must be set to the exact size of the attribute value. See `mtapi_actionattr_set()` for a list of predefined attribute numbers and the sizes of their values.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_actionattr_set()`

3.3.5 MTAPI_ACTION_GET_ATTRIBUTE

NAME

`mtapi_action_get_attribute()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_action_get_attribute(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns the attribute value that corresponds to the given `attribute_num` for this action.

`attribute` must point to the location where the attribute value is to be returned, and `attribute_size` must be set to the exact size of the attribute value. See `mtapi_actionattr_set()` for a list of predefined attribute numbers and the sizes of their values.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined.

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_actionattr_set()`

3.3.6 MTAPI_ACTION_DELETE

NAME

```
mtapi_action_delete()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_action_delete(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function deletes a software action. (Hardware actions exist perpetually and cannot be deleted.)

`mtapi_action_delete()`may be called by any node that has a valid action handle. Tasks associated with an action that has been deleted may still be executed depending on their internal state:

- If `mtapi_action_delete()` is called on an action that is currently executing, the associated task's state will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accomplish this, action functions must poll the task state with `mtapi_context_taskstate_get()`. A call to `mtapi_task_wait()` on the task executing this code will return the status set by `mtapi_context_status_set()`, or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are started or enqueued but waiting for execution by the MTAPI runtime when `mtapi_action_delete()` is called will not be executed any more if the deleted action is the only action associated with that task. A call to `mtapi_task_wait()` will return the status `MTAPI_ERR_ACTION_DELETED`.
- Tasks that are started or enqueued after deletion of the action will return `MTAPI_ERR_ACTION_INVALID` if the deleted action is the only action associated with that task.

Calling `mtapi_action_get()` on a deleted action will return `MTAPI_ERR_ACTION_INVALID` if all actions implementing the job had been deleted.

The function `mtapi_action_delete()` blocks until the corresponding action code is left by all tasks that are executing the code or until the timeout is reached. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function only returns `MTAPI_SUCCESS` if no tasks are executing the action when it is called. If it is set to `MTAPI_INFINITE`, the function may block infinitely.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_TIMEOUT`	Timeout was reached.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_taskstate_get()`
`mtapi_action_disable()`

3.3.7 MTAPI_ACTION_DISABLE

NAME

`mtapi_action_disable()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_action_disable(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function disables an action. Tasks associated with an action that has been disabled may still be executed depending on their internal state:

- If `mtapi_action_disable()` is called on an action that is currently executing, the associated task's state will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accomplish this, action functions must poll the task with `mtapi_context_taskstate_get()`. A call to `mtapi_task_wait()` on the task executing this code will return the status set by `mtapi_context_status_set()`, or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are started or enqueued but waiting for execution by the MTAPI runtime when `mtapi_action_disable()` is called will not be executed any more if the disabled action is the only action associated with that task. A call to `mtapi_task_wait()` will return the status `MTAPI_ERR_ACTION_DISABLED`.
- Tasks that are started or enqueued after the action has been disabled will return `MTAPI_ERR_ACTION_DISABLED` if either the disabled action is the only action associated with a task or all actions associated with a task are disabled.

`mtapi_action_disable()` blocks until all running tasks exit the code, or until the timeout is reached. If `timeout` is the constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function only returns `MTAPI_SUCCESS` if no tasks are executing the action when it is called. If it is set to `MTAPI_INFINITE` the function may block infinitely.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_TIMEOUT`	Timeout was reached.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_taskstate_get()`
`mtapi_action_delete()`

3.3.8 MTAPI_ACTION_ENABLE

NAME

```
mtapi_action_enable()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_action_enable(
  MTAPI_IN mtapi_action_hndl_t action,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function enables a previously disabled action. If this function is called on an action that no longer exists, an `MTAPI_ERR_ACTION_INVALID` error will be returned.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ACTION_INVALID`	Argument is not a valid action handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_taskstate_get()`

3.4 Action Functions and Action Context Functions

An action function is the executable software function that implements an action. It is invoked by the runtime with a signature as follows:

```
typedef void (*mtapi_action_function_t)(
    void* args,                            /* arguments */
    mtapi_size_t args_size,                /* length of arguments */
    void* result_buffer,                   /* buffer for storing results */
    mtapi_size_t result_buffer_size,       /* length of result_buffer */
    void* node_local_data,                 /* node-local data, shared by
                                           /* several tasks, executed */
                                           /* on the same node */
    mtapi_size_t node_local_data_size,     /* length of shared data */
    mtapi_task_context_t * context         /* MTAPI task context provided
                                              by the runtime systems
                                              identifying the current task
                                              for calling back the runtime
                                              system from the action
                                              function */
);
```

The runtime passes arguments to the action function when a task is started. Passing arguments from one node to another node should be implemented as a copy operation. Just as the arguments are passed before start of execution, the result buffer is copied back to the calling node after the action function terminates. In shared memory environments, the copying of data in both cases is not necessary. The node-local data is data used by several action functions being executed on the same node (or at least in the same address space). The shared data is specified when the action is created.

An action function can interact with the runtime environment through a task context object of type `mtapi_task_context_t` . A task context object is allocated and managed by the runtime. The runtime passes a pointer to the context object when the action function is invoked. The action may then query information about the execution context (e.g., its core number, the number of tasks and task number in a multi-instance task, polling the task state) by calling the `mtapi_context_*` functions. Furthermore it is possible to pass information from the action function to the runtime system which is executing the action function (setting the status manually, for example). All of these `mtapi_context_*` functions are called in the context of task execution.

3.4.1 MTAPI_CONTEXT_STATUS_SET

NAME

```
mtapi_context_status_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_context_status_set(
  MTAPI_INOUT mtapi_task_context_t* task_context,
  MTAPI_IN mtapi_status_t error_code,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function can be called from an action function to set the status that can be obtained by a subsequent call to `mtapi_task_wait()` or `mtapi_group_wait_any()`.

`task_context` must be the same value as the `context` parameter that the runtime passes to the action function when it is invoked.

The status can be passed from the action function to the runtime system by setting `error_code` to one of the following values:

- `MTAPI_SUCCESS` for successful completion
- `MTAPI_ERR_ACTION_CANCELLED` if the action execution is cancelled
- `MTAPI_ERR_ACTION_FAILED` if the task could not be completed as intended

The error code will be especially important in future versions of MTAPI where tasks shall be chained (flow graphs). The chain execution can then be aborted if the error code is not `MTAPI_SUCCESS`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_task_wait()`
`mtapi_group_wait_any()`

3.4.2 MTAPI_CONTEXT_RUNTIME_NOTIFY

NAME

```
mtapi_context_runtime_notify()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_context_runtime_notify(
  MTAPI_IN mtapi_task_context_t* task_context,
  MTAPI_IN mtapi_notification_t notification,
  MTAPI_IN void* data,
  MTAPI_IN mtapi_size_t data_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function can be called from an action function to notify the runtime system. This is used to communicate certain states to the runtime implementation to allow it to optimize task execution.

`task_context` must be the same value as the `context` parameter that the runtime passes to the action function when it is invoked.

The underlying type `mtapi_notification_t` and the valid values for `notification` are implementation-defined. The notification system is meant to be flexible, and can be used in many ways, for example:

- To trigger prefetching of data for further processing
- To order execution via queues there might be point in the action code where the next task in the queue may be started, even if the current code, started from the same queue, is still executing

RETURN VALUE

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

ERRORS

MTAPI_ERR_CONTEXT_OUTOFCONTEXT	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

SEE ALSO

`mtapi_task_wait()`
`mtapi_group_wait_any()`

3.4.3 MTAPI_CONTEXT_TASKSTATE_GET

NAME

```
mtapi_context_taskstate_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_task_state_t mtapi_context_taskstate_get(
  MTAPI_IN mtapi_task_context_t* task_context,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

An action function may call this function to obtain the state of the task that is associated with the action function.

`task_context` must be the same value as the `context` parameter that the runtime passes to the action function when it is invoked.

The underlying representation of type `mtapi_task_state_t` is implementation-defined. Values of type `mtapi_task_state_t` may be copied, assigned, and compared with other values of type `mtapi_task_state_t`, but the caller should make no other assumptions about its type or contents. A minimal implementation must return a status of `MTAPI_TASK_CANCELLED` if the task is cancelled, and `MTAPI_TASK_RUNNING` otherwise. Other values of the task state are implementation-defined. This task state can be used to abort a long running computation inside an action function.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime system.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

3.4.4 MTAPI_CONTEXT_INSTNUM_GET

NAME

`mtapi_context_instnum_get()`

SYNOPSIS

```
#include <mtapi.h>

mtapi_uint_t mtapi_context_instnum_get(
  MTAPI_IN mtapi_task_context_t* task_context,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function can be called from an action function to query the instance number of the associated task. A task can have multiple instances (multi-instance tasks), in which case the same job is executed multiple times in parallel. Each instance has a number, and this function gives the instance number. Task instances are numbered sequentially, starting at zero.

`task_context` must be the same value as the `context` parameter that the runtime passes to the action function when it is invoked.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the task instance number is returned. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_numinst_get()`

3.4.5 MTAPI_CONTEXT_NUMINST_GET

NAME

`mtapi_context_numinst_get()`

SYNOPSIS

```
#include <mtapi.h>

mtapi_uint_t mtapi_context_numinst_get(
  MTAPI_IN mtapi_task_context_t* task_context,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function can be called from an action function to query the total number of parallel task instances. This value is greater than one for multi-instance tasks.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_numinst_get()`

3.4.6 MTAPI_CONTEXT_CORENUM_GET

NAME

```
mtapi_context_corenum_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_uint_t mtapi_context_corenum_get(
  MTAPI_IN mtapi_task_context_t* task_context,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function can be called from an action function to query the current core number for debugging purposes. The core numbering is implementation-defined.

`task_context` must be the same value as the `context` parameter that the runtime passes to the action function when it was invoked.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the core number is returned. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_CONTEXT_OUTOFCONTEXT`	Not called in the context of a task execution. This function must be used in an action function only. The action function must be called from the MTAPI runtime.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_numinst_get()`

3.5 Core Affinities

The action functions associated with a job can be restricted to execute on a subset of processor cores of a node. Core affinities define these constraints for actions.

To set core affinities, the application must allocate an affinity mask object of type `mtapi_affinity_t` and initialize it with a call to `mtapi_affinity_init()`. Affinities are specified by calling `mtapi_affinity_set()`. The application must also allocate and initialize an action attributes object of type `mtapi_action_attributes_t`. The affinity mask object is then passed to `mtapi_actionattr_set()` to set the prescribed affinities in the action attributes object. The action attributes object is then passed to `mtapi_action_create()` to create a new action with those attributes.

It is in the nature of core affinities to be highly hardware dependent. The least common denominator for different architectures is enabling and disabling core numbers in the affinity mask. Action-to-core affinities can be set via the action attribute `MTAPI_ACTION_AFFINITY` during the creation of an action.

3.5.1 MTAPI_AFFINITY_INIT

NAME

```
mtapi_affinity_init()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_affinity_init(
  MTAPI_OUT mtapi_affinity_t* mask,
  MTAPI_IN mtapi_boolean_t affinity,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function initializes an affinity mask object. The affinity to all cores will be initialized to the value of `affinity`. This function should be called prior to calling `mtapi_affinity_set()` to specify non-default affinity settings. The affininity mask object may then be used to set the `MTAPI_ACTION_AFFINITY` attribute when creating an action with `mtapi_action_create()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_AFFINITY_MASK`	Invalid `mask` parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_affinity_set()`
`mtapi_action_create()`

3.5.2 MTAPI_AFFINITY_SET

NAME

`mtapi_affinity_set()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_affinity_set(
  MTAPI_INOUT mtapi_affinity_t* mask,
  MTAPI_IN mtapi_uint_t core_num,
  MTAPI_IN mtapi_boolean_t affinity,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function is used to change the default values of an affinity mask object. The affinity mask object can then be passed to `mtapi_actionattr_set()` to set the `MTAPI_ACTION_AFFINITY` action attribute. An action function will be executed on a core only if the core's affinity is set to `MTAPI_TRUE.` Calls to `mtapi_affinity_set()` have no effect on action attributes after the action has been created.

`mask` must be a pointer to an affinity mask object previously initialized with `mtapi_affinity_init()`.

The `core_num` is a hardware- and implementation-specific numeric identifier for a single core of the current node.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_AFFINITY_MASK`	Invalid `mask` parameter.
`MTAPI_ERR_CORE_NUM`	Unknown core number.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_affinity_init()`
`mtapi_actionattr_set()`

3.5.3 MTAPI_AFFINITY_GET

NAME

```
mtapi_affinity_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_boolean_t mtapi_affinity_get(
  MTAPI_IN mtapi_affinity_t* mask,
  MTAPI_IN mtapi_uint_t core_num,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns the affinity that corresponds to the given `core_num` for this affinity mask.

`mask` is a pointer to an affinity mask object previously initialized with `mtapi_affinity_init()`.

Note that affinities may be queried but may not be changed for an action after it has been created. If affinities need to be modified at runtime, new actions must be created.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the affinity for `core_num` is returned. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_AFFINITY_MASK`	Invalid `mask` parameter.
`MTAPI_ERR_CORE_NUM`	Unknown core number.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_affinity_init()`

3.6 Queues

Queues were made explicit in MTAPI. This allows mapping of queues onto hardware queues, if available. One MTAPI queue is associated with one action, or for purposes of load balancing, with actions implementing the same job on different nodes.

Queues are used to control the scheduling policy of tasks. The default scheduling policy for queues is ordered task execution. Tasks that have to be executed sequentially are enqueued into the same queue. In this case every queue is associated with exactly one action. Tasks started via different queues can be executed in parallel. This is needed for packet processing applications, for example: each stream is processed by one queue. This ensures sequential processing of packets belonging to the same stream. Different streams are processed in parallel.

3.6.1 MTAPI_QUEUEATTR_INIT

NAME

```
mtapi_queueattr_init()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queueattr_init(
  MTAPI_OUT mtapi_queue_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function initializes a queue attributes object. A queue attributes object is a container of queue attributes, optionally passed to `mtapi_queue_create()` to create a queue with non-default attributes.

The application is responsible for allocating the `mtapi_queue_attributes_t` object and initializing it with a call to `mtapi_queueattr_init()`. The application may then call `mtapi_queueattr_set()` to specify queue attribute values. Calls to `mtapi_queueattr_init()` have no effect on queue attributes after the queue has been created. To change an attribute of an existing queue, see `mtapi_queue_set_attribute()`. The `mtapi_queue_attributes_t` object may safely be deleted by the application after the call to `mtapi_queue_create()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_queueattr_set()`
`mtapi_queue_set_attribute()`
`mtapi_queue_create()`

3.6.2 MTAPI_QUEUEATTR_SET

NAME

```
mtapi_queueattr_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queueattr_set(
  MTAPI_INOUT mtapi_queue_attributes_t* attributes,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function sets queue attribute values in a queue attributes object. A queue attributes object is a container of queue attributes, optionally passed to `mtapi_queue_create()` to create a queue with non-default attributes.

`attributes` must be a pointer to a queue attributes object previously initialized by `mtapi_queueattr_init()`.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

Calls to `mtapi_queueattr_set()` have no effect on queue attributes once the queue has been created. The `mtapi_queue_attributes_t` object may safely be deleted by the application after the call to `mtapi_queue_create()`.

MTAPI-defined queue attributes:

Attribute num	Description	Data Type	Default
`MTAPI_QUEUE_GLOBAL`	Indicates if this is a globally visible queue. Only global queues are shared with other nodes.	`mtapi_boolean_t`	`MTAPI_TRUE`
`MTAPI_QUEUE_PRIORITY`	Priority of the queue.	`mtapi_uint_t`	`0` (default priority)
`MTAPI_QUEUE_LIMIT`	Max. number of elements in the queue; the queue blocks on queuing more items.	`mtapi_uint_t`	`0` (0 stands for ‘unlimited’)
`MTAPI_QUEUE_ORDERED`	Specify if the queue is order-preserving.	`mtapi_boolean_t`	`MTAPI_TRUE`
`MTAPI_QUEUE_RETAIN`	Allow enqueueing of jobs when queue is disabled.	`mtapi_boolean_t`	`MTAPI_FALSE`
`MTAPI_DOMAIN_SHARED`	Indicates if the queue is shareable across domains.	`mtapi_boolean_t`	`MTAPI_TRUE`

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_queueattr_init()`
`mtapi_queue_create()`

3.6.3 MTAPI_QUEUE_CREATE

NAME

```
mtapi_queue_create()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_queue_hndl_t mtapi_queue_create(
  MTAPI_IN mtapi_queue_id_t queue_id,
  MTAPI_IN mtapi_job_hndl_t job,
  MTAPI_IN mtapi_queue_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function creates a software queue object and associates it with the specified job. A job is associated with one or more actions that provide the executable implementation of the job. Hardware queues are considered to be pre-existant and do not need to be created.

`queue_id` is an identifier of implementation-defined type that must be supplied by the application. If `queue_id` is set to `MTAPI_QUEUE_ID_NONE`, the queue will be accessible only on the node on which it was created by using the returned queue handle. Otherwise the application may supply a `queue_id` by which the queue can be referenced domain-wide using `mtapi_queue_get()` to convert the id into a handle. The minimum and maximum values for `queue_id` may be derived from `MTAPI_MIN_USER_QUEUE_ID` and `MTAPI_MAX_USER_QUEUE_ID`.

`job` is a handle to a job obtained by a previous call to `mtapi_job_get()`. If `attributes` is `MTAPI_NULL`, the queue will be created with default attribute values. Otherwise `attributes` must point to a queue attributes object previously prepared using `mtapi_queueattr_init()` and `mtapi_queueattr_set()`.

There is an implementation-defined maximum number of queues permitted.

If more than one action is associated with the job, the runtime system chooses dynamically which action is used for execution (for load balancing purposes).

RETURN VALUE

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

ERRORS

`MTAPI_ERR_QUEUE_INVALID`	The `queue_id` is not a valid queue id.
`MTAPI_ERR_QUEUE_EXISTS`	This queue is already created.
`MTAPI_ERR_QUEUE_LIMIT`	Exceeded maximum number of queues allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_JOB_INVALID`	The associated job is not valid.

SEE ALSO

```
mtapi_job_get()
mtapi_queueattr_init()
mtapi_queueattr_set()
```

3.6.4 MTAPI_QUEUE_SET_ATTRIBUTE

NAME

`mtapi_queue_set_attribute()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queue_set_attribute(
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Changes the attribute value that corresponds to the given `attribute_num` for the specified queue.

See `mtapi_queueattr_set()` for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined.

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_queueattr_set()`

3.6.5 MTAPI_QUEUE_GET_ATTRIBUTE

NAME

```
mtapi_queue_get_attribute()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queue_get_attribute (
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns the attribute value that corresponds to the given `attribute_num` for the specified queue.

`attribute` must point to a location in memory sufficiently large to hold the returned attribute value. See `mtapi_queueattr_set()` for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and the `*attribute` value is undefined. If this function is called on a queue that no longer exists, an `MTAPI_ERR_QUEUE_INVALID` error will be returned.

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_queueattr_set()`

3.6.6 MTAPI_QUEUE_GET

NAME

```
mtapi_queue_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_queue_hndl_t mtapi_queue_get(
  MTAPI_IN mtapi_queue_id_t queue_id,
  MTAPI_IN mtapi_domain_t domain_id,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function converts a domain-wide `queue_id` into a node-local queue handle.

`queue_id` must match the `queue_id` that was associated with a software queue that was created with `mtapi_queue_create()`, or it must be a valid predefined queue identifier known a priori to the runtime and application (e.g., to reference a hardware queue. The minimum and maximum values for `queue_id` may be derived from `MTAPI_MIN_USER_QUEUE_ID` and `MTAPI_MAX_USER_QUEUE_ID`.

RETURN VALUE

On success, the queue handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. If this function is called on a queue that no longer exists, an `MTAPI_ERR_QUEUE_INVALID` error will be returned.

ERRORS

`MTAPI_ERR_QUEUE_INVALID`	The `queue_id` parameter does not refer to a valid queue or it is set to `MTAPI_QUEUE_ID_ANY`.
`MTAPI_ERR_NODE_NOTINIT`	The node/domain is not initialized.
`MTAPI_ERR_DOMAIN_NOTSHARED`	This resource cannot be shared by this domain.

SEE ALSO

`mtapi_queue_create()`

3.6.7 MTAPI_QUEUE_DELETE

NAME

`mtapi_queue_delete()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queue_delete(
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function deletes the specified software queue. Hardware queues are perpetual and cannot be deleted.

`queue` must be a valid handle to an existing queue.

`timeout` determines how long the function should wait for tasks already started via that queue to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function deletes the queue and returns immediately. If `timeout` is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

This function can be called from any node that has a valid queue handle. Tasks previously enqueued in a queue that has been deleted may still be executed depending on their internal state:

- If `mtapi_queue_delete()` is called on a queue that is currently executing an action, the task state of the corresponding task will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accomplish this, the action function must poll the task state with `mtapi_context_taskstate_get()`. A call to `mtapi_task_wait()` on the task executing this code will return the status set by `mtapi_context_status_set()`, or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are enqueued and waiting for execution by the MTAPI runtime environment when `mtapi_queue_delete()` is called will not be executed any more. A call to `mtapi_task_wait()` will return the status `MTAPI_ERR_QUEUE_DELETED`.
- Tasks that are enqueued after deletion of the queue will return a status of `MTAPI_ERR_QUEUE_INVALID`.

If this function is called on a queue that no longer exists, an `MTAPI_ERR_QUEUE_INVALID` status will be returned. A call to `mtapi_queue_get()` on a deleted queue will return `MTAPI_ERR_QUEUE_INVALID` as well, as long as no new queue has been created for the same queue ID.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_TIMEOUT`	Timeout was reached.

SEE ALSO

`mtapi_context_taskstate_get()`
`mtapi_task_wait()`

3.6.8 MTAPI_QUEUE_DISABLE

NAME

```
mtapi_queue_disable()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queue_disable(
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function disables the specified queue in such a way that it can be resumed later. This is needed to perform certain maintenance tasks. It can be called by any node that has a valid queue handle.

`timeout` determines how long the function should wait for tasks already started via that queue to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function deletes the queue and returns immediately. If timeout is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

Tasks previously enqueued in a queue that has been disabled may still be executed depending on their internal state:

- If `mtapi_queue_disable()` is called on a queue that is currently executing an action, the task state of the corresponding task will be set to `MTAPI_TASK_CANCELLED` and execution will continue. To accoomplish this, the action function must poll the task state by calling `mtapi_context_taskstate_get()`. A call to `mtapi_task_wait()` on the task executing this code will return the status set by `mtapi_context_status_set()`, or `MTAPI_SUCCESS` if not explicitly set.
- Tasks that are enqueued and waiting for execution by the MTAPI runtime environment when `mtapi_queue_disable()` is called will not be executed any more. They will be held in anticipation the queue is enabled again if the `MTAPI_QUEUE_RETAIN` attribute is set to `MTAPI_TRUE`. A call to `mtapi_task_wait()` will return the status `MTAPI_ERR_QUEUE_DISABLED`.
- Tasks that are enqueued after the queue had been disabled will return `MTAPI_ERR_QUEUE_DISABLED` if the `MTAPI_QUEUE_RETAIN` attribute is set to `MTAPI_FALSE`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_TIMEOUT`	Timeout was reached.

SEE ALSO

`mtapi_context_taskstate_get()`
`mtapi_task_wait()`

3.6.9 MTAPI_QUEUE_ENABLE

NAME

```
mtapi_queue_enable()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_queue_enable(
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function may be called from any node with a valid queue handle to re-enable a queue previously disabled with `mtapi_queue_disable()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_queue_disable()`

3.7 Jobs

A task is a particular invocation of a job. A job refers to one or more actions. An action is a hardware or software implementation of a job. In some cases, an action is referenced by an action handle, while in other cases, an action is referenced indirectly through a job handle. Each job is represented by a domain-wide job ID, or by a job handle which is local to one node.

Several actions can implement the same job based on different hardware resources (for instance a job can be implemented by one action on a DSP and by another action on a general purpose core, or a job can be implemented by both hardware and software actions).

3.7.1 MTAPI_JOB_GET

NAME

```
mtapi_job_get()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_job_hndl_t mtapi_job_get(
  MTAPI_IN mtapi_job_id_t job_id,
  MTAPI_IN mtapi_domain_t domain_id,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Given a `job_id`, this function returns the MTAPI handle for referencing the actions implementing the job. This function converts a domain-wide job ID into a node-local job handle.

RETURN VALUE

On success, the action handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_JOB_INVALID`	The `job_id` parameter does not refer to a valid action.
`MTAPI_ERR_DOMAIN_NOTSHARED`	This resource cannot be shared by this domain.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

3.8 Tasks

A task represents a piece of work “in flight” (similar to a thread handle). A task is associated with a job object, which is associated with one or more actions implementing the same job for load balancing purposes.

A task may optionally be associated with a task group. A task has attributes, and an internal state. A task begins its lifetime with a call to `mtapi_task_start()` or `mtapi_task_enqueue()`. A task is referenced by a handle of type `mtapi_task_hndl_t`. The underlying type of `mtapi_task_hndl_t` is implementation defined. Task handles may be copied, assigned, and passed as arguments, but otherwise the application should make no assumptions about the internal representation of a task handle.

Once a task is started, it is possible to wait for task completion from other parts of the program.

3.8.1 MTAPI_TASKATTR_INIT

NAME

```
    mtapi_taskattr_init()
```

SYNOPSIS

```
    #include <mtapi.h>

    void mtapi_taskattr_init(
      MTAPI_OUT mtapi_task_attributes_t* attributes,
      MTAPI_OUT mtapi_status_t* status
   );
```

DESCRIPTION

This function initializes a task attributes object. A task attributes object is a container of task attributes. It is an optional argument passed to `mtapi_task_start()` or `mtapi_task_enqueue()` to specify non-default task attributes when starting a task.

To set task attributes to non-default values, the application must allocate a task attributes object of type `mtapi_task_attributes_t` and initialize it with a call to `mtapi_taskattr_init()`. The application may call `mtapi_taskattr_set()` to specify attribute values. Calls to `mtapi_taskattr_init()` have no effect on task attributes after the task has started. The `mtapi_task_attributes_t` object may safely be deleted by the application after the task has started.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_task_start()`
`mtapi_task_enqueue()`

3.8.2 MTAPI_TASKATTR_SET

NAME

```
mtapi_taskattr_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_taskattr_set(
  MTAPI_INOUT mtapi_task_attributes_t* attributes,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_IN void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function sets task attribute values in a task attributes object. A task attributes object is a container of task attributes, optionally passed to `mtapi_task_start()` or `mtapi_task_enqueue()` to specify non-default task attributes when starting a task.

`attributes` is a pointer to a task attributes object that was previously initialized with a call to `mtapi_taskattr_init()`. Calls to `mtapi_taskattr_set()` have no effect on task attributes after the task has been created. The task attributes object may safely be deleted by the application after the task has started.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

MTAPI-defined task attributes:

Attribute num	Description	Data Type	Default
`MTAPI_TASK_DETACHED`	Indicates if this is a detached task. A detached task is deleted by MTAPI runtime after execution. The task handle of detached tasks must not be used, i.e., it is not possible to wait for completion of dedicated detached tasks. But it is possible to add detached tasks to a group and wait for completion of the group.	`mtapi_boolean_t`	`MTAPI_FALSE`
`MTAPI_TASK_INSTANCES`	Indicates how many parallel instances of task shall be started by MTAPI. The default case is that each task is executed exactly once. Setting this value to *n*, the corresponding action code will be executed *n* times, in parallel, if the underlying hardware allows it. (see chapter 4.1.7 Multi-Instance Tasks, page 107)	`mtapi_uint_t`	1

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_task_start()`
`mtapi_task_enqueue()`

3.8.3 MTAPI_TASK_START

NAME

```
mtapi_task_start()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_task_hndl_t mtapi_task_start(
  MTAPI_IN mtapi_task_id_t task_id,
  MTAPI_IN mtapi_job_hndl_t job,
  MTAPI_IN void* arguments,
  MTAPI_IN mtapi_size_t arguments_size,
  MTAPI_OUT void* result_buffer,
  MTAPI_IN mtapi_size_t result_size,
  MTAPI_IN mtapi_task_attributes_t* attributes,
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function schedules a task for execution. A task is associated with a job. A job is associated with one or more actions. An action provides an action function, which is the executable implementation of a job. If more than one action is associated with the job, the runtime system chooses dynamically which action is used for execution for load balancing purposes.

`task_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_TASK_ID_NONE`. The minimum and maximum values for `task_id` may be derived from `MTAPI_MIN_USER_TASK_ID` and `MTAPI_MAX_USER_TASK_ID`.

`job` must be a handle to a job obtained by a previous call to `mtapi_job_get()`.

If `arguments_size` is not zero, then `arguments` must point to data of `arguments_size` bytes. The arguments will be transferred by the runtime from the node where the action was created to the executing node if necessary. Marshalling of arguments is not part of the MTAPI specification and is implementation-defined.

If `attributes` is `MTAPI_NULL`, the task will be started with default attribute values. Otherwise `attributes` must point to a task attributes object previously prepared using `mtapi_taskattr_init()` and `mtapi_taskattr_set()`. The attributes of a task cannot be changed after the task is created.

`group` must be set to `MTAPI_GROUP_NONE` if the task is not part of a task group. Otherwise group must be a group handle obtained by a previous call to `mtapi_group_create()`.

RETURN VALUE

On success, a task handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_TASK_LIMIT`	Exceeded maximum number of tasks allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid group handle.
`MTAPI_ERR_JOB_INVALID`	The associated job is not valid.

SEE ALSO

`mtapi_taskattr_init()`
`mtapi_job_get()`
`mtapi_taskattr_set()`
`mtapi_group_create()`

3.8.4 MTAPI_TASK_ENQUEUE

NAME

```
mtapi_task_enqueue()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_task_hndl_t mtapi_task_enqueue(
  MTAPI_IN mtapi_task_id_t task_id,
  MTAPI_IN mtapi_queue_hndl_t queue,
  MTAPI_IN void* arguments,
  MTAPI_IN mtapi_size_t arguments_size,
  MTAPI_OUT void* result_buffer,
  MTAPI_IN mtapi_size_t result_size,
  MTAPI_IN mtapi_task_attributes_t* attributes,
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function schedules a task for execution using a queue. A queue is a task associated with a job. A job is associated with one or more actions. An action provides an action function, which is the executable implementation of a job.

`task_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_TASK_ID_NONE`. The underlying type of `mtapi_task_id_t` is implementation-defined. The minimum and maximum values for `task_id` may be derived from `MTAPI_MIN_USER_TASK_ID` and `MTAPI_MAX_USER_TASK_ID`.

`queue` must be a handle to a queue obtained by a previous call to `mtapi_queue_create()`.

If `arguments_size` is not zero, then `arguments` must point to data of `arguments_size` bytes. The arguments will be transferred by the runtime from the node where the action was created to the executing node. Marshalling of arguments is not part of the MTAPI specification and is implementation-defined.

If `attributes` is `MTAPI_NULL`, the task will be started with default attribute values. Otherwise `attributes` must point to a task attributes object previously prepared using `mtapi_taskattr_init()` and `mtapi_taskattr_set()`. Once a task has been enqueued, its attributes may not be changed.

`group` must be set to `MTAPI_GROUP_NONE` if the task is not part of a task group. Otherwise `group` must be a group handle obtained by a previous call to `mtapi_group_create()`.

RETURN VALUE

On success, a task handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_TASK_LIMIT`	Exceeded maximum number of tasks allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_QUEUE_INVALID`	Argument is not a valid queue handle.

SEE ALSO

`mtapi_taskattr_init()`
`mtapi_taskattr_set()`

3.8.5 MTAPI_TASK_GET_ATTRIBUTE

NAME

`mtapi_task_get_attribute()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_task_get_attribute(
  MTAPI_IN mtapi_task_hndl_t task,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns a copy of the attribute value that corresponds to the given `attribute_num` for the specified task. The attribute value will be returned in `*attribute`. Note that task attributes may be queried but may not be changed after a task has been created.

`task` must be a valid handle to a task that was obtained by a previous call to `mtapi_task_start()` or `mtapi_task_enqueue()`.

See `mtapi_task_set_attribute()` for a list of predefined attribute numbers and the sizes of the attribute values. The application is responsible for allocating sufficient space for the returned attribute value and for setting `attribute_size` to the exact size in bytes of the attribute value.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and the attribute value is undefined. If this function is called on a task that no longer exists, an `MTAPI_ERR_TASK_INVALID` error code will be returned.

ERRORS

MTAPI_ERR_PARAMETER	Invalid `attribute` parameter.
MTAPI_ERR_TASK_INVALID	Argument is not a valid task handle.
MTAPI_ERR_ATTR_NUM	Unknown attribute number.
MTAPI_ERR_ATTR_SIZE	Incorrect attribute size.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.

SEE ALSO

`mtapi_taskattr_set()`

3.8.6 MTAPI_TASK_CANCEL

NAME

`mtapi_task_cancel()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_task_cancel(
  MTAPI_IN mtapi_task_hndl_t task,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function cancels a task and sets the task status to `MTAPI_TASK_CANCELLED`.

`task` must be a valid handle to a task that was obtained by a previous call to `mtapi_task_start()` or `mtapi_task_enqueue()`.

If the execution of a task has not been started, the runtime system might remove the task from the runtime-internal task queues. If task execution is already running, an action function implemented in software can poll the task status and react accordingly.

Since the task is referenced by a task handle which can only be used node-locally, a task can be cancelled only on the node where the task was created.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_TASK_INVALID`	Argument is not a valid task handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_task_start()`
`mtapi_task_enqueue()`

3.8.7 MTAPI_TASK_WAIT

NAME

```
mtapi_task_wait()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_task_wait(
  MTAPI_IN mtapi_task_hndl_t task,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function waits for the completion of the specified task.

`task` must be a valid handle to a task that was obtained by a previous call to `mtapi_task_start()` or `mtapi_task_enqueue()`. The task handle becomes invalid on a successful wait, i.e., after the task had run to completion and `mtapi_task_wait()` returns `MTAPI_SUCCESS`.

`timeout` determines how long the function should wait for tasks already started via that queue to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function does not block and returns immediately. If timeout is set to `MTAPI_INFINITE` the function may block infinitely. Other values for timeout and the units of measure are implementation-defined.

Results of completed tasks can be obtained via `result_buffer` associated with the task. The size of the buffer has to be equal to the result size written in the action code. If the result is not needed by the calling code, `result_buffer` may be set to `MTAPI_NULL`. For multi-instance tasks, the result buffer is filled by an array of all the task instances' results. I.e., the result buffer has to be allocated big enough (number of instances times size of result).

Calling `mtapi_task_wait()` more than once for the same task results in undefined behavior.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below. If this function is called on a task that no longer exists, an `MTAPI_ERR_TASK_INVALID` error code will be returned. Status will be `MTAPI_ERR_ARG_SIZE` or `MTAPI_ERR_RESULT_SIZE` if the sizes of arguments or result buffer do not match.

ERRORS

`MTAPI_ERR_TASK_INVALID`	Argument is not a valid task handle.
`MTAPI_TIMEOUT`	Timeout was reached.
`MTAPI_ERR_PARAMETER`	Invalid `timeout` parameter.
`MTAPI_ERR_TASK_CANCELLED`	The task has been cancelled because of `mtapi_task_cancel()` was called before the task was executed or the error code was set to `MTAPI_ERR_TASK_CANCELLED` by `mtapi_context_status_set()` in the action function.
`MTAPI_ERR_WAIT_PENDING`	`mtapi_task_wait()` had already been called for the same task and the first wait call is still pending.

`MTAPI_ERR_ACTION_CANCELLED`	Action execution was cancelled by the action function (`mtapi_context_status_set()`).
`MTAPI_ERR_ACTION_FAILED`	Error set by action function (`mtapi_context_status_set()`).
`MTAPI_ERR_ACTION_DELETED`	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action function to `MTAPI_ERR_ACTION_DELETED` by `mtapi_context_status_set()`.
`MTAPI_ERR_ARG_SIZE`	The size of the arguments expected by action differs from arguments size of the caller.
`MTAPI_ERR_RESULT_SIZE`	The size of the result buffer expected by action differs from result buffer size of the caller.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_context_status_set()`
`mtapi_task_start()`
`mtapi_task_enqueue()`

3.9 Task Groups

Task groups allow synchronizing on a group of tasks. This concept is similar to barrier synchronization of threads. MTAPI specifies a minimal task group feature set in order to allow small and efficient implementations.

3.9.1 MTAPI_GROUPATTR_INIT

NAME

`mtapi_groupattr_init()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_groupattr_init(
  MTAPI_OUT mtapi_group_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function initializes a group attributes object. A group attributes object is a container of group attributes. It is an optional argument passed to `mtapi_group_create()` to specify non-default group attributes when creating a task group.

To set group attributes to non-default values, the application must allocate a group attributes object of type `mtapi_group_attributes_t` and initialize it with a call to `mtapi_groupattr_init()`. The application may call `mtapi_groupattr_set()` to specify attribute values. Calls to `mtapi_groupattr_init()` have no effect on group attributes after the group has been created. The `mtapi_group_attributes_t` object may safely be deleted by the application after the call to `mtapi_group_create()`.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_groupattr_set()`
`mtapi_group_create()`

3.9.2 MTAPI_GROUPATTR_SET

NAME

```
mtapi_groupattr_set()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_groupattr_set(
     MTAPI_INOUT mtapi_group_attributes_t* attributes,
     MTAPI_IN mtapi_uint_t attribute_num,
     MTAPI_IN void* attribute,
     MTAPI_IN mtapi_size_t attribute_size,
     MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function sets group attribute values in a group attributes object. A group attributes object is a container of group attributes, optionally passed to `mtapi_group_create()` to specify non-default group attributes when creating a task group.

`attributes` is a pointer to a group attributes object that was previously initialized with a call to `mtapi_groupattr_init()`. Calls to `mtapi_groupattr_set()` have no effect on group attributes after the group has been created. The group attributes object may safely be deleted by the application after the call to `mtapi_group_create()`.

See the table below for a list of predefined attribute numbers and the sizes of the attribute values. The application must set `attribute_size` to the exact size in bytes of the attribute value. Additional attributes may be defined by the implementation.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_ATTR_READONLY`	Attribute cannot be modified.
`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_groupattr_init()`
`mtapi_group_create()`

3.9.3 MTAPI_GROUP_CREATE

NAME

```
mtapi_group_create()
```

SYNOPSIS

```
#include <mtapi.h>

mtapi_group_hndl_t mtapi_group_create(
  MTAPI_IN mtapi_group_id_t group_id,
  MTAPI_IN mtapi_group_attributes_t* attributes,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function creates a task group and returns a handle to the group. After a group is created, a task may be associated with a group when the task is started with `mtapi_task_start()` or `mtapi_task_enqueue()`.

`group_id` is an optional ID provided by the application for debugging purposes. If not needed, it can be set to `MTAPI_GROUP_ID_NONE`. The underlying type of `mtapi_group_id_t` is implementation-defined. The minimum and maximum values for `group_id` may be derived from `MTAPI_MIN_USER_GROUP_ID` and `MTAPI_MAX_USER_GROUP_ID`.

If `attributes` is `MTAPI_NULL`, the group will be created with default attribute values. Otherwise `attributes` must point to a group attributes object previously prepared using `mtapi_groupattr_init()` and `mtapi_groupattr_set()`.

RETURN VALUE

On success, a group handle is returned and `*status` is set to `MTAPI_SUCCESS`. On error, `*status` is set to the appropriate error defined below.

ERRORS

`MTAPI_ERR_GROUP_LIMIT`	Exceeded maximum number of groups allowed.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.
`MTAPI_ERR_PARAMETER`	Invalid `attributes` parameter.

SEE ALSO

`mtapi_groupattr_init()`
`mtapi_groupattr_set()`
`mtapi_task_start()`
`mtapi_task_enqueue()`

3.9.4 MTAPI_GROUP_SET_ATTRIBUTE

NAME

`mtapi_group_set_attribute()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_group_set_attribute (
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Changes the value of the attribute that corresponds to the given `attribute_num` for the specified task group.

`attribute` must point to the attribute value, and `attribute_size` must be set to the exact size of the attribute value. See `mtapi_groupattr_set()` for a list of predefined attribute numbers and the sizes of their values.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid group handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_groupattr_init()`
`mtapi_groupattr_set()`

3.9.5 MTAPI_GROUP_GET_ATTRIBUTE

NAME

```
mtapi_group_get_attribute()
```

SYNOPSIS

```
#include <mtapi.h>

void mtapi_group_get_attribute (
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_IN mtapi_uint_t attribute_num,
  MTAPI_OUT void* attribute,
  MTAPI_IN mtapi_size_t attribute_size,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

Returns the attribute value that corresponds to the given `attribute_num` for this task group.

`attribute` must point to the location where the attribute value is to be returned, and `attribute_size` must be set to the exact size of the attribute value. See `mtapi_groupattr_set()` for a list of predefined attribute numbers and the sizes of their values.

RETURN VALUE

On success, `*status` is set to `MTAPI_SUCCESS` and the attribute value is returned in `*attribute`. On error, `*status` is set to the appropriate error defined below and `*attribute` is undefined. If this function is called on a group that no longer exists, an `MTAPI_ERR_GROUP_INVALID` error code will be returned.

ERRORS

`MTAPI_ERR_PARAMETER`	Invalid `attribute` parameter.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid group handle.
`MTAPI_ERR_ATTR_NUM`	Unknown attribute number.
`MTAPI_ERR_ATTR_SIZE`	Incorrect attribute size.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_groupattr_set()`

3.9.6 MTAPI_GROUP_WAIT_ALL

NAME

`mtapi_group_wait_all()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_group_wait_all(
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function waits for the completion of a task group. Tasks may be associated with groups when the tasks are started. Each task is associated with one or more actions. This function returns when all the associated action functions have completed or cancelled. The group handle becomes invalid if this function returns `MTAPI_SUCCESS`.

`timeout` determines how long the function should wait for tasks already started in the group to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function does not block and returns immediately. If `timeout` is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

To obtain results from a task, the application should call `mtapi_group_wait_any()`instead.

During execution, an action function may optionally call `mtapi_context_status_set()` to set a task status that will be returned in this function in `*status`. If multiple action functions set different task status values, it is implementation-defined which of those is returned in `mtapi_group_wait_all()`. The following task status values may be set by an action function: `MTAPI_ERR_TASK_CANCELLED`, `MTAPI_ERR_ACTION_CANCELLED`, `MTAPI_ERR_ACTION_FAILED`, and `MTAPI_ERR_ACTION_DELETED`.

RETURN VALUE

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

ERRORS

`MTAPI_TIMEOUT`	Timeout was reached.
`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid task handle.
`MTAPI_ERR_WAIT_PENDING`	`mtapi_group_wait_all()` had already been called for the same group and the first wait call is still pending.
`MTAPI_ERR_PARAMETER`	Invalid `timeout` parameter.
`MTAPI_ERR_ARG_SIZE`	The size of the arguments expected by action differs from arguments size of the caller.
`MTAPI_ERR_RESULT_SIZE`	The size of the result buffer expected by action differs from result buffer size of the caller.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

`MTAPI_GROUP_COMPLETED`	Group completed, i.e., there are no more task to wait for in the group.
`MTAPI_ERR_TASK_CANCELLED`	At least one task has been cancelled because of `mtapi_task_cancel()` was called before the task was executed or the error code was set to `MTAPI_ERR_TASK_CANCELLED` by `mtapi_context_status_set()` in the action function.
`MTAPI_ERR_ACTION_CANCELLED`	The action execution of at least one task was cancelled by the action function (`mtapi_context_status_set()`).
`MTAPI_ERR_ACTION_FAILED`	Error set by at least one action function (`mtapi_context_status_set()`).
`MTAPI_ERR_ACTION_DELETED`	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action function to `MTAPI_ERR_ACTION_DELETED` by `mtapi_context_status_set()`.

SEE ALSO

`mtapi_context_status_set()`
`mtapi_group_wait_any()`

3.9.7 MTAPI_GROUP_WAIT_ANY

NAME

`mtapi_group_wait_any()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_group_wait_any(
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_OUT void** result,
  MTAPI_IN mtapi_timeout_t timeout,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function waits for the completion of any task in a task group. Tasks may be associated with groups when the tasks are started. Each task is associated with one or more actions. This function returns when any of the associated action functions have completed or have been cancelled.

The group handle does not become invalid if this function returns `MTAPI_SUCCESS`. The group handle becomes invalid if this function returns `MTAPI_GROUP_COMPLETED`.

`group` must be a valid group handle obtained by a previous call to `mtapi_group_create()`.

Action functions may pass results that will be available in `*result` after `mtapi_group_wait_any()` returns. If the results are not needed, `result` may be set to `MTAPI_NULL`. Otherwise, `result` must point to an area in memory of sufficient size to hold the array of results from the completed task(s). The size of the result buffer is given in the argument `result_buffer_size` that the runtime passes to an action function upon invocation.

`timeout` determines how long the function should wait for a task in the group to finish. The underlying type of `mtapi_timeout_t` is implementation-defined. If `timeout` is a constant 0 or the symbolic constant `MTAPI_NOWAIT`, this function does not block and returns immediately. If `timeout` is set to `MTAPI_INFINITE` the function may block infinitely. Other values for `timeout` and the units of measure are implementation defined.

During execution, an action function may optionally call `mtapi_context_status_set()` to set a task status that will be returned in this function in `*status`. The following task status values may be set by an action function: `MTAPI_ERR_TASK_CANCELLED`, `MTAPI_ERR_ACTION_CANCELLED`, `MTAPI_ERR_ACTION_FAILED`, and `MTAPI_ERR_ACTION_DELETED`.

RETURN VALUE

On success, `*status` is either set to `MTAPI_SUCCESS` if one of the tasks in the group completed or to `MTAPI_GROUP_COMPLETED` if all tasks of the group have completed and successfully waited for. On error, `*status` is set to the appropriate error defined below.

ERRORS

MTAPI_TIMEOUT	Timeout was reached.
MTAPI_ERR_GROUP_INVALID	Argument is not a valid task handle.
MTAPI_ERR_PARAMETER	Invalid timeout parameter.
MTAPI_ERR_ARG_SIZE	The size of the arguments expected by action differs from arguments size of the caller.
MTAPI_ERR_RESULT_SIZE	The size of the result buffer expected by action differs from result buffer size of the caller.
MTAPI_ERR_NODE_NOTINIT	The calling node is not initialized.
MTAPI_GROUP_COMPLETED	Group completed, i.e., there are no more tasks to wait for in the group.
MTAPI_ERR_TASK_CANCELLED	The task has been cancelled because `mtapi_task_cancel()` was called before the task was executed, or the error code was set to `MTAPI_ERR_TASK_CANCELLED` by `mtapi_context_status_set()` in the action code.
MTAPI_ERR_ACTION_CANCELLED	Action execution was cancelled by the action function (`mtapi_context_status_set()`).
MTAPI_ERR_ACTION_FAILED	Error set by action function (`mtapi_context_status_set()`).
MTAPI_ERR_ACTION_DELETED	All actions associated with the task have been deleted before the execution of the task was started or the error code has been set in the action code to `MTAPI_ERR_ACTION_DELETED` by `mtapi_context_status_set()`.

SEE ALSO

`mtapi_context_status_set()`
`mtapi_group_wait_all()`

3.9.8 MTAPI_GROUP_DELETE

NAME

`mtapi_group_delete()`

SYNOPSIS

```
#include <mtapi.h>

void mtapi_group_delete(
  MTAPI_IN mtapi_group_hndl_t group,
  MTAPI_OUT mtapi_status_t* status
);
```

DESCRIPTION

This function deletes a task group. Deleting a group does not have any influence on tasks belonging to the group. Adding tasks to a group that is already deleted will result in an `MTAPI_ERR_GROUP_INVALID` error.

RETURN VALUE

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

ERRORS

`MTAPI_ERR_GROUP_INVALID`	Argument is not a valid group handle.
`MTAPI_ERR_NODE_NOTINIT`	The calling node is not initialized.

SEE ALSO

`mtapi_group_create()`

4. FAQ

In order to keep the examples short, the status returned by the MTAPI function is checked by the macro `MTAPI_CHECK_STATUS()`. This macro may terminate the program, for example.

4.1 Basic Examples

4.1.1 Initialize MTAPI

The MTAPI runtime has to be initialized on every node. The domain and node IDs must be defined in a global header file, e.g.:

```
/* IDs to be defined in a global header file */
#define THIS_DOMAIN_ID 01
#define THIS_NODE_ID   01
```

The following snippet shows an example for the initialization:

```
mtapi_node_attributes_t node_attr;
mtapi_info_t info;
mtapi_status_t status;

/* initialize attributes */
mtapi_nodeattr_init(&node_attr, &status);
MTAPI_CHECK_STATUS(status);

/* modify parameters (if needed) */
mtapi_nodeattr_set(
    &node_attr,
    MTAPI_NODE_TYPE,       // example attribute
    MTAPI_NODE_TYPE_SMP,   // example attribute value
    MTAPI_NODE_TYPE_SIZE,  // example attribute size
    &status
);
MTAPI_CHECK_STATUS(status);

/* initialize MTAPI node */
mtapi_initialize(
    THIS_DOMAIN_ID,
    THIS_NODE_ID,
    &node_attr,
    &info,
    &status
);
MTAPI_CHECK_STATUS(status);
```

4.1.2 Coordinating Tasks on a Single Node

If the application code and all tasks shared the same memory (which is the case on SMP multicore processors), actions and jobs are created during an initialization phase and later used during runtime:

```
mtapi_status_t status;
mtapi_action_hndl_t action;
mtapi_job_hndl_t job;
mtapi_task_hndl_t task;

int args = 42;
int args_size = sizeof(int);

/* create action and jobs (initialization phase) */

/* initializing action attributes */
mtapi_actionattr_init(&action_attributes, &status);
MTAPI_CHECK_STATUS(status);

/* create action */
action = mtapi_action_create(
    JOB01,                              /* action ID, defined by the appl. */
    (exampleActionFunction),            /* action function */
    MTAPI_NULL,                         /* no shared data */
    0,                                  /* length of shared data */
    //MTAPI_DEFAULT_ACTION_ATTRIBUTES,    /* action attributes */
    &action_attributes,                 /* action attributes */
    &status                             /* status out-parameter */
);
MTAPI_CHECK_STATUS(status);

/* create job */
job = mtapi_job_get(JOB01, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);

/* work with tasks at runtime */

/* start task */
task = mtapi_task_start(
    MTAPI_TASK_ID_NONE,                 /* optional task ID */
    job,                                /* job */
    (void*)&args,                       /* arguments passed to action functions
*/
    args_size,                          /* size of arguments */
    MTAPI_NULL,                               /* result buffer */
    0,                                  /* size of result buffer */
    MTAPI_DEFAULT_TASK_ATTRIBUTES,      /* task attributes */
    MTAPI_GROUP_NONE,                   /* optional task group */
    &status                             /* status out-parameter */
);
MTAPI_CHECK_STATUS(status);

/* do something else */
doSomethingElse();

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);
```

4.1.3 Implementing Jobs on Different Nodes

Working with different nodes (e.g., when actions are implemented on different processors), is very similar to the single node example.

Suppose we have two nodes, where node A is a general purpose processor (for example a quad-core shared memory architecture), and node B is a special purpose processor, not sharing memory with node A. Our application shall be executed on node A. One job, called “compute” in this example, shall be executed by node B.

First we initialize the MTAPI runtime on both nodes (`mtapi_initialize()`). On node B we have to write the code needed for the computation:

On node B we create an action object for the computation and register it at the runtime:

```
mtapi_action_create(EXAMPLE_GLOBAL_JOB_ID_4711, (someActionFunction),
             MTAPI_NULL, 0, MTAPI_DEFAULT_ATTRIBUTES, &status);
MTAPI_CHECK_STATUS(status);
```

Now, on the coordination node (A) we can obtain a handle of the action object:

```
/* get job handle */
job = mtapi_job_get(JOB_4711, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);

/* start task  */
task = mtapi_task_start(MTAPI_TASK_ID_NONE, job, (void*)&args, args_size,
MTAPI_NULL, 0, MTAPI_DEFAULT_TASK_ATTRIBUTES, MTAPI_GROUP_NONE, &status);
MTAPI_CHECK_STATUS(status);

/* do something else */
doSomethingElse();

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);
```

4.1.4 Returning Task Results

The action function associated with a task may return a task result. When a task is started the caller passes a result buffer. If the corresponding action code is executed on the same node, it directly copies the results into that buffer. Results from remote actions are copied to the buffer on the caller side by the MTAPI runtime system.

```
/* user-defined result type */
result_example_t result;

/* start task passing the address of the result buffer */
task = mtapi_task_start(
        MTAPI_TASK_ID_NONE,                /* optional task ID */
        job,                               /* job */
        (void*)&args,                      /* arguments passed to action func’s */
        args_size,                         /* size of arguments */
        &result,                           /* result buffer */
        sizeof(result_example_t),          /* size of result buffer */
        MTAPI_DEFAULT_TASK_ATTRIBUTES,     /* task attributes */
        MTAPI_GROUP_NONE,                  /* optional task group */
        &status                            /* status out-parameter */
);
MTAPI_CHECK_STATUS(status);

/* do something else */
doSomethingElse();

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);

/* work with result ... */
```

We pass a buffer for the result (`result`) and the expected size (in this case `sizeof(result_example_t)`). If the result, written by the action has the wrong size, `status` after `mtapi_task_wait()` will be `MTAPI_ERR_RESULT_SIZE`.

The corresponding code in the action implementation looks like this:

```
void exampleActionFunction(
    void* args, mtapi_size_t arg_size,
    void* result_buffer, mtapi_size_t result_buffer_size,
    void* node_local_data, mtapi_size_t node_local_data_size,
    mtapi_task_context_t* task_context)
{
    mtapi_status_t status;
    int i, argument;
    result_example_t* result;
    result_example_t local_result;

    printf("exampleActionFunction() called\n");

    /**** prepare arguments ****/
    /* check size of arg's (in this case we only expect one integer value) */
    if (arg_size != sizeof(int)) {
        printf("wrong size of arguments\n");
        mtapi_context_status_set(task_context, MTAPI_ERR_ARG_SIZE, &status);
        MTAPI_CHECK_STATUS(status);
        return;
    }
    /* cast arguments to the desired type */
    argument = *(int*)args;

    /**** set result buffer ****/
    /* if the caller is not interested in results, result_buffer may be
       MTAPI_NULL. Of course, this depends on the application */
    if (result_buffer == MTAPI_NULL) {
        result = &local_result; /* write the results somewhere else (example) */
    } else {
        /* if results are expected by the caller, check result buffer size... */
        if (result_buffer_size == sizeof(result_example_t)) {
            /* ... and cast the result buffer */
            result = (result_example_t*)result_buffer;
        } else {
            printf("wrong size of result buffer\n");
            mtapi_context_status_set(
                task_context,
                MTAPI_ERR_RESULT_SIZE,
                &status
            );
            MTAPI_CHECK_STATUS(status);
            return;
        }
    }

    /**** some calculation ****/
    for(i = 0; i < 10; i++) {
        printf("task %i working\n", argument);
    }

    /**** write results  ****/
    result->value1 = 47;
    result->value2 = argument;

    return;
}
```

4.1.5 Using Task Groups

The purpose of task groups is to synchronize with a group of tasks. Task groups are designed to allow efficient handling of thousands of tasks in one group. This leads to some restrictions:

- Tasks cannot belong to more than one group.
- Waiting for tasks is the only functional API for groups. It is not possible to cancel all tasks in a group or trigger anything else influencing all tasks of a group via the group API (the API is designed to support a very simple group implementation: tasks are internally referencing the group, but task groups do not have to maintain a list of tasks belonging to them).

There are two ways of synchronizing with groups:

- `mtapi_group_wait_all()`
  blocks until all tasks belonging to a particular group have finished, or the timeout has expired.
- `mtapi_group_wait_any()`
  blocks until one of the tasks in the group has finished, or the timeout has expired.

Example for `mtapi_group_wait_all()`**:**

```
mtapi_status_t status; mtapi_action_hndl_t action; mtapi_job_hndl_t job;
mtapi_task_attributes_t task_attributes; mtapi_group_hndl_t group;
int argument[NUM_TASKS], i; const mtapi_size_t args_size = sizeof(int);
const mtapi_boolean_t att_val_true = MTAPI_TRUE;
const mtapi_size_t bool_size = sizeof(mtapi_boolean_t);

/* create action */
action = mtapi_action_create(MY_JOB_01, (exampleActionFunction), MTAPI_NULL, 0,
MTAPI_DEFAULT_ACTION_ATTRIBUTES, &status);
MTAPI_CHECK_STATUS(status);

/* get job */
job = mtapi_job_get(JOB01, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);

/* set task attribute DETACHED because we are not interested in task handles
after the tasks have been started */
mtapi_taskattr_init (&task_attributes, &status);
mtapi_taskattr_set (&task_attributes, MTAPI_TASK_DETACHED,(void*)&att_val_true,
bool_size, &status);

/* prepare group */
group = mtapi_group_create(MTAPI_GROUP_ID_NONE, MTAPI_DEFAULT_GROUP_ATTRIBUTES,
&status);

/* create several tasks using the same group (in this example we use the same
   action for all of them, of course it is possible to use different actions for
   the different tasks) */
for (i = 0; i < 10; i++) {
    /* start task */
    mtapi_task_start(MTAPI_TASK_ID_NONE, job, (void*)&argument[i],
      args_size, MTAPI_NULL, 0, &task_attributes, group, &status);
    MTAPI_CHECK_STATUS(status);
}

/* do something else */
doSomethingElse();

/* wait for completion of all tasks in the group */
mtapi_group_wait_all(group, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);
```

Example for `mtapi_group_wait_any()`**:**

In order to show processing of results, we allocate a buffer for the results of all tasks (the `results` array) and pass one array element to each of the tasks at `mtapi_task_start()`. `mtapi_group_wait_any()` provides the result buffer of each completed task as an output parameter. This makes processing of results easy.

```
mtapi_status_t status;
mtapi_action_hndl_t action;
mtapi_job_hndl_t job;
mtapi_task_attributes_t task_attributes;
mtapi_group_hndl_t group;
int argument[NUM_TASKS], i;
int args_size = sizeof(int);
const mtapi_boolean_t att_val_true = MTAPI_TRUE;
const mtapi_size_t bool_size = sizeof(mtapi_boolean_t);
result_example_t results[NUM_TASKS];
result_example_t* tmp_result;


/* ... create action, set task attributes and prepare group as above ...*/

/* create several tasks using the same group */
for (i = 0; i < NUM_TASKS; i++) {
    argument[i] = i;   /* since we pass a pointer make sure that the value
                          is at the same address until task completed working */
    /* start task */
    mtapi_task_start(MTAPI_TASK_ID_NONE, job, (void*)&argument[i],
      args_size, &results[i], sizeof(result_example_t), &task_attributes,
      group, &status);
    MTAPI_CHECK_STATUS(status);
}


/* do something else */
doSomethingElse();

/* wait for completion of tasks in the group and handle results */
while (TRUE) {
    mtapi_group_wait_any(group, (void**)&tmp_result, MTAPI_INFINITE,  &status);
    /* status will be MTAPI_ERR_RESULT_SIZE on result size mismatch */
    if (status != MTAPI_SUCCESS) {  /* MTAPI_GROUP_COMPLETED */
        break;
    }
    /* ... process 'tmp_result' here ...
       temp_result contains the pointer that was passed at mtapi_task_start
       to the task which just returned */
    printf("result.value1 = %i, result.value2 = %i\n",
      tmp_result->value1, tmp_result->value2);
}
mtapi_group_delete(group, &status);
MTAPI_CHECK_STATUS(status);
```

4.1.6 Task Cancellation

Tasks can be cancelled before they finish execution. There are two possible scenarios:

- A task may be started (i.e., scheduled for execution by the runtime system), but execution was not started. In this case the runtime system might remove the task from the runtime-internal task queues.

- Task execution is already running. If the task is implemented in software, the task status may be polled by the action code (cooperative cancellation).

Tasks can be cancelled only on the node that created a task (it holds the task handle). On this node a task is cancelled by calling `mtapi_task_cancel(task, &status);`

The corresponding code in the action implementation looks like this:

```
void exampleActionFunction(
    void* args, mtapi_size_t arg_size,
    void* result_buffer, mtapi_size_t result_buffer_size,
    void* node_local_data, mtapi_size_t node_local_data_size,
    mtapi_task_context_t* task_context)
{
    mtapi_status_t status;
    int i, argument;
    result_example_t* result;
    result_example_t local_result;

    printf("exampleActionFunction() called\n");

    /**** prepare arguments ****/
    /* ... */

    /**** set result buffer ****/
    /* ... */

    /**** long running calculation that can be cancelled ****/
    for(i = 0; i < 10; i++) {
        if (mtapi_context_taskstate_get(task_context, &status) ==
                                                     MTAPI_TASK_CANCELLED) {
            printf("task cancelled\n");
            mtapi_context_status_set(
                task_context,
                MTAPI_ERR_ACTION_CANCELLED,
                &status
            );
            return;
        }
        printf("task %i working\n", argument);
        Sleep(100);
    }

    result->value1 = 47;
    result->value2 = argument;

    return;
}
```

4.1.7 Multi-Instance Tasks

Multi-instance tasks are tasks that execute the same action multiple times in parallel. Multi-instance tasks are started by setting the task attributes `MTAPI_TASK_INSTANCES` to a value greater than one.

An action function can determine the current instance number with `mtapi_context_instnum_get()` and the total number of instances executed in parallel with `mtapi_context_numinst_get()`.

In most cases an action function prepared for multi-instance tasks looks like this:

```
void multiInstanceActionFunction(
    void* args, mtapi_size_t arg_size,
    void* result_buffer, mtapi_size_t result_buffer_size,
    void* node_local_data, mtapi_size_t node_local_data_size,
    mtapi_task_context_t* task_context)
{
    mtapi_status_t status;
    mtapi_int_t this_instance, num_instances;
    result_example_t* result;

    printf("multiInstanceActionFunction() called\n");

    /* no arguments */

    /* check result buffer size... */
    if (result_buffer_size == sizeof(result_example_t)) {
        /* ... and cast the result buffer */
        result = (result_example_t*)result_buffer;
    } else {
        printf("wrong size of result buffer\n");
        mtapi_context_status_set(task_context, MTAPI_ERR_RESULT_SIZE, &status);
        MTAPI_CHECK_STATUS(status);
        return;
    }

    num_instances = mtapi_context_numinst_get(task_context, &status);
    this_instance = mtapi_context_instnum_get(task_context, &status);

    printf("task working, instance %i of %i\n", this_instance, num_instances);

    /* ... select data chunks to work with depending on this_instance
       and num_instances (lines or tiles of an image, for example) */

    /* ... compute result here ... */

    /* dummy for calculating result */
    result->value1 = this_instance;
    result->value2 = num_instances;

    //mtapi_context_status_set(task_context, MTAPI_SUCCESS, &status);
    MTAPI_CHECK_STATUS(status);
}
```

In order to retrieve the results, a result buffer has to be allocated before calling `mtapi_task_start()` for a multi-instance task. The results are copied, one after the other, into that array:

```
mtapi_status_t status;
mtapi_action_hndl_t action;
mtapi_job_hndl_t job;
mtapi_task_attributes_t task_attributes;
mtapi_task_hndl_t task;

mtapi_uint_t args = 42, i;
int args_size = sizeof(mtapi_uint_t);

const mtapi_uint_t instances = 5;
const mtapi_size_t inst_size = sizeof(mtapi_uint_t);

/* user-defined result type */
result_example_t *result_ptr;

/* create action */
action = mtapi_action_create(MY_JOB_02, (multiInstanceActionFunction),
MTAPI_NULL, 0, MTAPI_DEFAULT_ACTION_ATTRIBUTES, &status);
MTAPI_CHECK_STATUS(status);

/* get job */
job = mtapi_job_get(JOB02, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);

/* initialize task attributes and set MTAPI_TASK_INSTANCES */
mtapi_taskattr_init (&task_attributes, &status);
MTAPI_CHECK_STATUS(status);

mtapi_taskattr_set(&task_attributes, MTAPI_TASK_INSTANCES,
                       (void*)&instances, inst_size, &status);
MTAPI_CHECK_STATUS(status);

/* allocate buffer for results (we skipped malloc error handling for brevity) */
result_ptr = (result_example_t *) malloc(instances * sizeof(result_example_t));

/* start task */
task = mtapi_task_start(
    MTAPI_TASK_ID_NONE,                   /* no task ID */
    job,                                  /* job */
    (void*)&args, args_size,              /* arguments */
    result_ptr, sizeof(result_example_t), /* result buffer for all instances */
    &task_attributes,                     /* task attributes */
    MTAPI_GROUP_NONE,                     /* task does not belong to group */
    &status                               /* status output parameter */
);
MTAPI_CHECK_STATUS(status);

doSomethingElse();   /* do something else */

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);

/* process results */
for (i=0; i<instances; i++) {
    printf("result %i: value1 = %i, value2 = %i\n",
                            i, result_ptr[i].value1, result_ptr[i].value2);
}
free(result_ptr);
```

4.1.8 Load Balancing

More than one action can be associated with one job before a task is started (`mtapi_task_start()`) or a queue is created (`mtapi_queue_create()`). The runtime system selects one of the actions when execution is triggered either by `mtapi_task_start()` or by `mtapi_task_enqueue()`.

Possible implementations (this has to be defined by the implementation):

a) Round robin

b) Dependent on the queue fill level of the queues on the nodes where the actions are located

c) One queue on the coordinating node; whenever a node idles it requests new work from other nodes.

All actions associated with the same job shall implement the same piece of work. The intention is to have not more than one implementation for a job per node (however, the API would also allow two actions implemented on the same node to be associated with the same job).

Example:

```
mtapi_status_t status;
mtapi_job_hndl_t job;
mtapi_task_attributes_t task_attributes;
mtapi_task_hndl_t task;
int args = 42;
int args_size = sizeof(int);

/*
On the remote node
action = mtapi_action_create(JOB01, (exampleActionFunction), MTAPI_NULL, 0,
MTAPI_DEFAULT_ACTION_ATTRIBUTES, &status);
has to called for software implemented actions implementing the same job;
Jobs implmented in hardware have actions IDs predefined by the MTAPI
implementation and can be
used the same way as remote actions.
*/

/* get job */
job = mtapi_job_get(JOB01, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);

/* initialize task attributes */
mtapi_taskattr_init(&task_attributes, &status);
MTAPI_CHECK_STATUS(status);

/* start task  */
task = mtapi_task_start(
    MTAPI_TASK_ID_NONE,      /* no task ID for this example */
    job,                     /* associated with MORE THAN ONE action */
    (void*)&args, args_size, /* arguments */
    MTAPI_NULL, 0,                 /* no result buffer for this example */
    &task_attributes,        /* task attributes */
    MTAPI_GROUP_NONE,        /* task does not belong to group */
    &status                  /* status output parameter */
);
MTAPI_CHECK_STATUS(status);

/* do something else */
doSomethingElse();

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
```

```
MTAPI_CHECK_STATUS(status);
```

4.1.9 Task Queues

Using queues is similar to starting tasks. Just before `mtapi_task_enqueue()` can be called, a queue has to be created:

```
mtapi_status_t status;
mtapi_action_hndl_t action;
mtapi_job_hndl_t job;
mtapi_task_hndl_t task;
mtapi_queue_hndl_t queue;

int args = 42;
int args_size = sizeof(int);

/* create action (for remote tasks use mtapi_action_get instead) */
action = mtapi_action_create(MY_JOB_01, (exampleActionFunction),
                         MTAPI_NULL, 0, MTAPI_DEFAULT_ACTION_ATTRIBUTES,
&status);
MTAPI_CHECK_STATUS(status);

/* get job */
job = mtapi_job_get(JOB01, THIS_DOMAIN_ID, &status);
MTAPI_CHECK_STATUS(status);


/* create queue */
queue = mtapi_queue_create(
    MY_QUEUE_01,                    /* queue ID */
    job,                            /* associated with one action
                                       in this example */
    MTAPI_DEFAULT_QUEUE_ATTRIBUTES, /* task attributes */
    &status                         /* status output parameter */
);
MTAPI_CHECK_STATUS(status);

/* enqueue task in queue (repeat this for all tasks that should be processed by
that queue) */
task = mtapi_task_enqueue(
    MTAPI_TASK_ID_NONE,            /* no task ID for this example */
    queue,                         /* QUEUE */
    (void*)&args, args_size,       /* arguments */
    MTAPI_NULL, 0,                       /* no result buffer for this example
*/
    MTAPI_DEFAULT_TASK_ATTRIBUTES, /* task attributes */
    MTAPI_GROUP_NONE,              /* task does not belong to group */
    &status                        /* status output parameter */
);
MTAPI_CHECK_STATUS(status);

/* do something else */
doSomethingElse();

/* wait for task completion */
mtapi_task_wait(task, MTAPI_INFINITE, &status);
MTAPI_CHECK_STATUS(status);
```

4.1.10 Setting an Action-to-Core Affinity

Setting a core affinity restricts an action to be executed on a subset of processor cores on the node where the action is executed. Example:

```
/* ... */

mtapi_affinity_t affinity;
mtapi_action_attributes_t action_attributes;

/* ... */

/* create affinity mask */
mtapi_affinity_init(&affinity, MTAPI_FALSE, &status); /* disable all cores */
MTAPI_CHECK_STATUS(status);

mtapi_affinity_set(&affinity, 3, MTAPI_TRUE, &status); /* enable core 3 */
MTAPI_CHECK_STATUS(status);

/* adding the affinity to action attributes */
mtapi_actionattr_init(&action_attributes, &status);
MTAPI_CHECK_STATUS(status);
mtapi_actionattr_set(&action_attributes, MTAPI_ACTION_AFFINITY, &affinity,
sizeof(mtapi_affinity_t), &status);
MTAPI_CHECK_STATUS(status);

/* create action */
action = mtapi_action_create(
    JOB01,                              /* action ID, defined by the appl. */
    (exampleActionFunction),            /* action function */
    MTAPI_NULL,                         /* no shared data */
    0,                                  /* length of shared data */
    &action_attributes,                 /* action attributes */
    &status                             /* status out-parameter */
);
MTAPI_CHECK_STATUS(status);
```

4.2 Porting Layer for Parallelization Libraries

MTAPI can be used as a porting layer for higher level parallel programming APIs providing libraries for data-parallel processing, data-flow graphs, or special purpose algorithms.

An implementation of the OpenMP runtime system, for example, can be made portable by using MTAPI as a foundation. If an optimized MTAPI implementation is available for a dedicated hardware, this even leads to better results compared to building the OpenMP runtime system on top of the operating system threads.

4.3 Event-driven Systems

4.3.1 Packet Processing

This example illustrates the use of MTAPI in a packet processing application. The hardware is set up to receive packet fragments from an interface and send them to reassembly queues. Then the software reassembles fragmented packets and processes reassembled packets. The hardware then receives processed packets from an output queue and sends them to an interface.

```
#include <mtapi.h>
#include <mtapi_extensions.h>
/* Vendor specific extensions to MTAPI. The following extensions are used in
this example:
   - void mtapi_dispatch(void)
     Allows MTAPI runtime to dispatch one task */

#define MY_DOMAIN_ID (...)
#define MY_NODE_ID (...)
#define MY_INIT_CORE_ID (...)

#define MY_REASSEMBLE_PACKET_JOB (...)
#define MY_PROCESS_PACKET_JOB (...)
#define MY_OUTPUT_PACKET_JOB (...)

#define MY_PRIORITY_NUM (...)
#define MY_FLOW_NUM (...)

#define MY_PACKET_HEADER_SIZE (...)

mtapi_queue_hndl_t my_reassembly_queue_hdl_tbl[MY_PRIORITY_NUM * MY_FLOW_NUM];
mtapi_queue_hndl_t my_process_queue_hdl_tbl[MY_PRIORITY_NUM];
mtapi_queue_hdl_t my_output_queue_hdl;

typedef struct my_packet_header_s {
      ...
} my_packet_header_t;

typedef struct my_flow_context_s {
      ...
} my_flow_context_t;

mtapi_boolean_t my_check_first(my_flow_context_t *context_ptr,
my_packet_header_t *header_ptr)
{
      /* Checks that header is first of packet. */
}

mtapi_boolean_t my_check_last(my_flow_context_t *context_ptr, my_packet_header_t
*header_ptr)
{
      /* Checks that header is last of packet. */
}

mtapi_boolean_t my_check_sanity(my_flow_context_t *context_ptr,
my_packet_header_t *header_ptr)
{
      /* Checks that header arrives in order and is consistent with context. */
}
```

```
mtapi_uint_t my_get_core_id(void)
{
      /* gets ID of this core */
}

mtapi_uint_t my_get_fragment_size(my_packet_header_t *header_ptr)
{
      /* reads fragment size from fragment header */
}

mtapi_uint8_t *my_get_packet_ptr(my_flow_context_t *context_ptr)
{
      /* reads packet pointer from context */
}

mtapi_uint_t my_get_packet_size(my_flow_context_t *context_ptr)
{
      /* reads packet size from context */
}

mtapi_uint_t my_get_output_packet_size(mtapi_uint_t input_packet_size)
{
      /* computes output packet size from input packet size */
}

mtapi_uint_t my_get_priority(my_flow_context_t *context_ptr)
{
      /* reads priority from context */
}

mtapi_uint8_t *my_get_write_ptr(my_flow_context_t *context_ptr)
{
      /* reads write pointer from context */
}

void my_init_header(my_flow_context_t *context_ptr, my_packet_header_t
*header_ptr)
{
      /* initializes packet header based on context */
}

void my_set_packet_ptr(my_flow_context_t *flow_context_ptr, mtapi_uint8_t
*packet_ptr)
{
      /* writes packet pointer to flow context */
}

void my_set_priority(my_flow_context_t *flow_context_ptr, mtapi_uint_t priority)
{
      /* writes priority to flow context */
}

void my_set_write_ptr(my_flow_context_t *flow_context_ptr, mtapi_uint8_t
*packet_ptr)
{
      /* writes write pointer to flow context */
}

void my_start_output_hw(void)
{
      /* starts HW that will read packets from output queue,
         write packets to interface */
}
```

```
void my_start_input_hw(mtapi_uint_t priority, mtapi_uint_t flow_id)
{
      /* starts HW that will read packets from interface,
         classify packets according to priority and flow ID,
         write packets to corresponding reassembly queue */
}

void my_create_reassembly_queue(
      mtapi_uint_t priority,
      mtapi_uint_t flow_id)
{
      mtapi_status_t status;
      mtapi_action_hndl_t action_hdl,
      mtapi_job_hndl_t job_hdl;
      mtapi_queue_attributes_t* queue_attr_ptr;
      my_flow_context_t *flow_context_ptr;

      flow_context_ptr = (my_flow_context_t *)malloc(sizeof(my_flow_context_t));
      my_set_priority(flow_context_ptr, priority);
      action_hdl = mtapi_action_create(
            MY_REASSEMBLE_PACKET_JOB, my_reassemble_packet,
            (void *)flow_context_ptr, sizeof(my_flow_context_t),
            MTAPI_DEFAULT_ACTION_ATTRIBUTES,
            &status);
      MTAPI_CHECK_STATUS(status);

      job_hdl = mtapi_job_get(MY_REASSEMBLE_PACKET_JOB, MY_DOMAIN_ID, &status);
      MTAPI_CHECK_STATUS(status);

      mtapi_queueattr_init (queue_attr_ptr, &status);
      MTAPI_CHECK_STATUS(status);

      mtapi_queueattr_set (
            queue_attr_ptr,
            MTAPI_QUEUE_ORDERED, (void *)MTAPI_TRUE, MTAPI_QUEUE_ORDERED_SIZE,
            &status);
      MTAPI_CHECK_STATUS(status);

      mtapi_queueattr_set (
            queue_attr_ptr,
            MTAPI_QUEUE_PRIORITY, (void *)priority, MTAPI_QUEUE_PRIORITY_SIZE,
            &status);
      MTAPI_CHECK_STATUS(status);

      my_reassembly_queue_hdl_tbl[priority * MY_FLOW_NUM + flow_id] = \
            mtapi_queue_create(
                  MTAPI_QUEUE_ID_NONE,
                  MY_REASSEMBLE_PACKET_JOB,
                  queue_attr_ptr,
                  status
            );
      MTAPI_CHECK_STATUS(status);
}

void my_create_process_queue(
      mtapi_uint_t priority)
{
      mtapi_status_t status;
      mtapi_action_hndl_t action_hdl,
      mtapi_job_hndl_t job_hdl;
      mtapi_queue_attributes_t* queue_attr_ptr;
```

```
        action_hdl = mtapi_action_create(
                MY_PROCESS_PACKET_JOB, my_process_packet, NULL, 0,
                MTAPI_DEFAULT_ACTION_ATTRIBUTES,
                &status);
        MTAPI_CHECK_STATUS(status);

        job_hdl = mtapi_job_get(MY_PROCESS_PACKET_JOB, MY_DOMAIN_ID, &status);
        MTAPI_CHECK_STATUS(status);

        mtapi_queueattr_init (queue_attr_ptr, &status);
        MTAPI_CHECK_STATUS(status);

        mtapi_queueattr_set (
                queue_attr_ptr,
                MTAPI_QUEUE_PRIORITY, (void *)priority, MTAPI_QUEUE_PRIORITY_SIZE,
                &status
        );
        MTAPI_CHECK_STATUS(status);

        *queue_hdl_ptr = mtapi_queue_create(
                MTAPI_QUEUE_ID_NONE,
                MY_PROCESS_PACKET_JOB,
                queue_attr_ptr,
                status);
        MTAPI_CHECK_STATUS(status);
}

void my_create_output_queue(void)
{
        mtapi_status_t status;
        mtapi_job_hndl_t job_hdl;

        job_hdl = mtapi_job_get(MY_OUTPUT_PACKET_JOB, MY_DOMAIN_ID, &status);
        MTAPI_CHECK_STATUS(status);

        my_output_queue_hdl = mtapi_queue_create(
                MTAPI_QUEUE_ID_NONE,
                MY_OUTPUT_PACKET_JOB,
                MTAPI_NULL,
                status);
        MTAPI_CHECK_STATUS(status);
}

void my_reassemble_packet(
        void *args,
        mtapi_size_t args_size,
        void *,
        mtapi_size_t,
        void *node_local_data,
        mtapi_size_t,
        mtapi_task_context_t *)
{
        my_packet_header_t *fragment_header_ptr;
        my_flow_context_t *flow_context_ptr;

        fragment_header_ptr = (my_packet_header_t *)args;
        flow_context_ptr = (my_flow_context_t *)node_local_data;

        if (my_check_sanity(flow_context_ptr, fragment_header_ptr)) return;

        if (my_check_first(flow_context_ptr))
        {
                mtapi_size_t packet_size;
```

```
            mtapi_uint8_t *packet_ptr;
            my_packet_header_t *packet_header_ptr;

            packet_size = my_get_packet_size(flow_context_ptr);
            packet_ptr = (mtapi_uint8_t *)malloc(packet_size);
            packet_header_ptr = (my_packet_header_t *)packet_ptr;
            my_init_header(flow_context_ptr, packet_header_ptr);
            my_set_packet_ptr(flow_context_ptr, packet_ptr);
      }

      {
            mtapi_size_t payload_size;
            mtapi_uint8_t *payload_ptr, *write_ptr;

            payload_size = my_get_fragment_size(fragment_header_ptr) - \
                  MY_PACKET_HEADER_SIZE;
            payload_ptr = (uint8_t *)fragment_header_ptr + \
                  MY_PACKET_HEADER_SIZE;
            write_ptr = my_get_write_pointer(flow_context_ptr);
            memcpy(write_ptr, payload_ptr, payload_size);
            my_set_write_pointer(flow_context_ptr, write_ptr + \
                  fragment_payload_size);
      }

      if (my_check_last(flow_context_ptr))
      {
            mtapi_task_attributes *task_attr_ptr;
            mtapi_status_t status;
            mtapi_size_t packet_size;
            mtapi_uint_t priority;
            void *packet_ptr;

            mtapi_taskattr_init(task_attr_ptr, &status);
            MTAPI_CHECK_STATUS(status);

            mtapi_taskattr_set(
                  task_attr_ptr,
                  MTAPI_TASK_DETACHED,
                  (void *)MTAPI_TRUE,
                  MTAPI_TASK_DETACHED_SIZE,
                  &status);
            MTAPI_CHECK_STATUS(status);

            packet_ptr = my_get_packet_ptr(flow_context_ptr);
            packet_size = my_get_packet_size(flow_context_ptr);
            priority = my_get_priority(flow_context_ptr);

            mtapi_task_enqueue(
                  MTAPI_TASK_ID_NONE,
                  my_process_queue_hdl_tbl[priority],
                  packet_ptr, packet_size,
                  NULL, 0,
                  task_attr_ptr,
                  MTAPI_GROUP_ID_NONE,
                  &status);
            MTAPI_CHECK_STATUS(status);
      }
}

void my_process_packet(
      void *args,
      mtapi_size_t args_size,
      void *,
      mtapi_size_t,
      void *,
      mtapi_size_t,
      mtapi_task_context_t *)
{
      void *packet_ptr;
      mtapi_size_t packet_size;
      mtapi_status_t status;
      mtapi_task_attributes *task_attr_ptr;

      packet_size = my_get_output_packet_size(input_packet_size);
      packet_ptr = malloc(packet_size);
      my_process_data(args, args_size, packet_ptr, packet_size);

      mtapi_taskattr_init(task_attr_ptr, &status);
      MTAPI_CHECK_STATUS(status);

      mtapi_taskattr_set(
            task_attr_ptr,
            MTAPI_TASK_DETACHED, (void *)MTAPI_TRUE, MTAPI_TASK_DETACHED_SIZE,
            &status);
      MTAPI_CHECK_STATUS(status);

      mtapi_task_enqueue(
            MTAPI_TASK_ID_NONE,
            my_output_queue_hdl,
            packet_ptr, packet_size,
            NULL, 0,
            task_attr_ptr,
            MTAPI_GROUP_ID_NONE,
            &status);
      MTAPI_CHECK_STATUS(status);
}

int main(void)
{
      if (my_get_core_id() == MY_INIT_CORE_ID)
      {
            mtapi_status_t status;
            mtapi_info_t info;

            mtapi_initialize(MY_DOMAIN_ID, MY_NODE_ID, MTAPI_NULL, \
                                                              &info, &status);

            my_create_output_queue();
            my_start_output_hw();

            for (priority = 0; priority < MY_PRIORITIES_NUM; priority++)
            {
                  my_create_process_queue(priority);

                  for (flow_id = 0; flow_id < MY_FLOW_NUM; flow_id++)
                  {
                          my_create_reassembly_queue(priority, flow_id);
                          my_start_input_hw(priority, flow_id);
                  }
            }
      }

      while(1) mtapi_dispatch();
}
```

4.4 Parallel Algorithms

4.4.1 Recursive Algorithms

Calculations of Fibonacci numbers is a simple example for a recursive algorithm that can easily be parallelized. A sequential version is:

```
int fib(int n)
{
    int x,y;
    if (n < 2)
        return n;
    else
    {
        x = fib(n - 1);
        y = fib(n - 2);
        return x + y;
    }
}

int fibonacci(int n)
{
    int n = 10;
    {
        printf("fib(%i) = %i\n", n, fib(n));
    }
}
```

This can be parallelized by spawning a task for one of the recursive calls (`fib(n - 1)`, for example). When doing this with MTAPI, we have to create an action function that represents `fib(int n)`.

```
mtapi_job_hndl_t fibonacciJob;

void fibonacciActionFunction(
    void* args, mtapi_size_t arg_size,
    void* result_buffer, mtapi_size_t result_buffer_size,
    void* node_local_data, mtapi_size_t node_local_data_size,
    mtapi_task_context_t* task_context)
{
    mtapi_task_hndl_t task;
    mtapi_status_t status;

    int n;
    int* result;
    int x, y, a, b;


    /* check size of arguments (in this case we only expect one int value) */
    if (arg_size != sizeof(int)) {
        printf("wrong size of arguments\n");
        mtapi_context_status_set(task_context, MTAPI_ERR_ARG_SIZE, &status);
        MTAPI_CHECK_STATUS(status);
        return;
    }

    /* cast arguments to the desired type */
    n = *(int*)args;
    /* if the caller is not interested in results, result_buffer may be
       MTAPI_NULL. Of course, this depends on the application */
    if (result_buffer == MTAPI_NULL) {
        mtapi_context_status_set(task_context, MTAPI_ERR_RESULT_SIZE, &status);
        MTAPI_CHECK_STATUS(status);
    } else {
        /* if results are expected by the caller, check result buffer size... */
        if (result_buffer_size == sizeof(int)) {
            /* ... and cast the result buffer */
            result = (int*)result_buffer;
        } else {
            printf("wrong size of result buffer\n");
            mtapi_context_status_set(task_context, MTAPI_ERR_RESULT_SIZE,
                                                                 &status);
            MTAPI_CHECK_STATUS(status);
            return;
        }
    }


    /* calculate */

    if (n < 2)
        *result = n;
    else
    {
        /* first recursive call spawned as task (x = fib(n - 1);) */
        a = n - 1;
        task = mtapi_task_start(
            MTAPI_TASK_ID_NONE,                /* optional task ID */
            fibonacciJob,                      /* job */
            (void*)&a,                         /* arguments passed to action
                                                  functions */
            sizeof(int),                       /* size of arguments */
            (void*)&x,                         /* result buffer */
            sizeof(int),                       /* size of result buffer */
            MTAPI_DEFAULT_TASK_ATTRIBUTES,     /* task attributes */
            MTAPI_GROUP_NONE,                  /* optional task group */
            &status                            /* status out-parameter */
        );
        MTAPI_CHECK_STATUS(status);

        /* second recursive call can be called directly (y = fib(n - 2);) */
        b = n - 2;
        fibonacciActionFunction(&b, sizeof(int), &y, sizeof(int),
                                               MTAPI_NULL, 0, task_context);

        mtapi_task_wait(task, MTAPI_INFINITE, &status);
        MTAPI_CHECK_STATUS(status);

        *result = x + y;
    }
    return;
}
```

The `fibonacci()` function looks like this:

```
int fibonacci(int n)
{
    mtapi_action_hndl_t fibonacciAction;
    mtapi_task_hndl_t task;
    mtapi_status_t status;

    int result;

    /* create action */
    fibonacciAction = mtapi_action_create(
        FOBONACCI_JOB,                      /* action ID, defined by the appl.*/
        (fibonacciActionFunction),          /* action function */
        MTAPI_NULL,                         /* no shared data */
        0,                                  /* length of shared data */
        MTAPI_DEFAULT_ACTION_ATTRIBUTES,    /* action attributes */
        &status                             /* status out-parameter */
    );
    MTAPI_CHECK_STATUS(status);

    /* get job */
    fibonacciJob = mtapi_job_get(FOBONACCI_JOB, THIS_DOMAIN_ID, &status);
    MTAPI_CHECK_STATUS(status);

    /* start task */
    task = mtapi_task_start(
        MTAPI_TASK_ID_NONE,                 /* optional task ID */
        fibonacciJob,                       /* job */
        (void*)&n,                          /* arguments passed to action
                                               functions */
        sizeof(int),                        /* size of arguments */
        (void*)&result,                     /* result buffer */
        sizeof(int),                        /* size of result buffer */
        MTAPI_DEFAULT_TASK_ATTRIBUTES,      /* task attributes */
        MTAPI_GROUP_NONE,                   /* optional task group */
        &status                             /* status out-parameter */
    );
    MTAPI_CHECK_STATUS(status);

    /* wait for task completion */
    mtapi_task_wait(task, MTAPI_INFINITE, &status);
    MTAPI_CHECK_STATUS(status);

    /* print result */
   printf("result: %i\n", result);
   return result;
}
```

4.4.2 Smith-Waterman Sequence Alignment

The Smith-Waterman sequence alignment algorithm is an example from the bioinformatics domain, commonly used for exact local and global sequence alignment. We discuss a wavefront algorithm-based implementation of this example. We consider this algorithm to primarily enable computation and communication on different processors of a given environment.

For this example, MTAPI is used to perform the following tasks:

1. Create tasks, possibly remote, to be implemented on a different node.
2. Start the tasks.
3. Create and define groups and associate the tasks wherever necessary.
4. Enable completion of all tasks belonging to a particular group.
5. Synchronize between several tasks.
6. Wait for tasks to complete and wait for tasks belonging to the same group to finish completion.

Characteristics

Data access

Biological database searches can take months or weeks to complete the execution due to the data dependencies in the algorithm, limiting the usage of these algorithms. Creating groups out of tasks that can be executed concurrently can lead to better data movement patterns in this algorithm.

Load Balance

Dynamic load balancing is applied across all computational devices available. This load balancing technique appears to be best suited for this algorithm. The user can decide how and which resources to run the computations on. Static scheduling may at times not be a good solution since it lacks the ability to automatically adapt to the application irregularity as well as the system heterogeneity.

Task Size

On a general note, fine-grained chunks can achieve better load balancing. On the other hand, there is a trade-off between workload of a single task and the overhead generated by the runtime system.

Synchronization

Synchronization between the tasks and their execution completion is essential to avoid conflicts or accessing incorrect data. Since this particular application follows the wavefront algorithm, it is required that certain tasks finish execution before the other tasks resume their operation.

Metrics

Gap Penalty

The gap penalty is used during sequence alignment. This metric contributes to the overall score of the alignments. Gaps are inserted in order to align the sequences to maximize similarity in the algorithm. They correspond to the insertion or deletion of a substring.

Code Example

```c
// MTAPI example:
// Porting a Smith-Waterman wave-front algorithm from OpenMP to MTAPI
//
// Remarks:
//  - no error handling included (mtapi_status_t not evaluated)
//  - attributes to be set for action, group, and task not specified
//
//

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

// MTAPI: define entry function for tasks
// (parallel activities have to be implemented as functions)

// define data structure for task parameters

typedef struct  {
    unsigned int np;        // firstprivate
    unsigned int mp;        // firstprivate
    unsigned int i;         // firstprivate
    unsigned int k;         // firstprivate
    unsigned int l;         // firstprivate
    int **h;                // shared
    int **e;                // shared
    int **f;                // shared
    char ** sequences;      // shared
    unsigned int open;      // shared
    unsigned int extension; // shared
} element_args;

// task entry function (signature according to MTAPI specification)
void calculate_element(
    void* parameters, mtapi_size_t param_size,
    void* result_buffer, mtapi_size_t result_buffer_size,
    void* node_local_data, mtapi_size_t node_local_data_size,
    mtapi_task_context_t* task_context)
{
    int a;                  // private
    int b;                  // private

    // get arguments (cast to concrete parameter struct)
    element_args* args = (element_args*) parameters;

    // optionally extract arguments (alternatively access directly further down)
    unsigned int np = args->np;
    unsigned int mp = args->mp;
    unsigned int i = args->i;
    unsigned int k = args->k;
    unsigned int l = args->l;
    int **h = args->h;
    int **e = args->e;
    int **f = args->f;
    char ** sequences = args->sequences;
    unsigned int open = args->open;
    unsigned int extension = args->extension;

    //printf("(%d,%d)", np-i, mp+i);
    // Calculate e
    a = h[(np-i)][(mp+i)-1] - open;
    b = e[(np-i)][(mp+i)-1] - extension;
    e[(np-i)][(mp+i)] = ((a>b)?a:b);

    // Calculate f
    a = h[(np-i)-1][(mp+i)] - open;
    b = f[(np-i)-1][(mp+i)] - extension;
    f[(np-i)][(mp+i)] = ((a>b)?a:b);

    // Calculate h
    a = h[(np-i)-1][(mp+i)-1] + similarity(sequences[k][(np-i)-1],
                                               sequences[l][(mp+i)-1]);
    if(e[(np-i)][(mp+i)] > f[(np-i)][(mp+i)]) {
        if(a > e[(np-i)][(mp+i)])     h[(np-i)][(mp+i)] = ((a>0)?a:0);
        else h[(np-i)][(mp+i)] = ((e[(np-i)][(mp+i)]>0)?e[(np-i)][(mp+i)]:0);
    }
    else {
        if(a > f[(np-i)][(mp+i)])     h[(np-i)][(mp+i)] = ((a>0)?a:0);
        else h[(np-i)][(mp+i)] = ((f[(np-i)][(mp+i)]>0)?f[(np-i)][(mp+i)]:0);
    }
}

// END of MTAPI task entry function


int similarity(char x, char y) {
    return((x==y)?2:-1);
}

int main(int argc, char *argv[]) {

    // MTAPI: initialize MTAPI

    mtapi_status_t mtapi_status;
    mtapi_info_t mtapi_info;
    mtapi_node_attributes_t mtapi_node_attributes;
    mtapi_action_attributes_t mtapi_action_attributes;
    mtapi_job_hndl_t job;
    mtapi_group_hndl_t group;

    mtapi_nodeattr_init (
        &mtapi_node_attributes,
        &mtapi_status
        );

    mtapi_nodeattr_set (
        &mtapi_node_attributes,
        MTAPI_NODE_TYPE,       // example attribute
        MTAPI_NODE_TYPE_SMP,   // example attribute value
        MTAPI_NODE_TYPE_SIZE,  // example attribute size
        &mtapi_status
        );

    // initialize MTAPI
    mtapi_initialize(THIS_DOMAIN_ID,THIS_NODE_ID,
                         &mtapi_node_attributes, &mtapi_info, &mtapi_status);

    // MTAPI: create action object
    mtapi_actionattr_init(
        &mtapi_action_attributes,
        &mtapi_status
    );

    mtapi_actionattr_set (
        &mtapi_action_attributes,
        // ...
    );

    mtapi_action_hndl_t action =  mtapi_action_create(
        WAVEFRONT_JOB_ID,     // to be defined in a central header file
        calculate_element,
        MTAPI_NULL, 0,
        &mtapi_action_attributes,
        &mtapi_status
    );

    // MTAPI: get job handle for starting a task
    // (job handles allow to start remote tasks, i.e., actions implemented on a
    // different node)
    job = mtapi_job_get(WAVEFRONT_JOB_ID, THIS_DOMAIN_ID, &status);

    // MTAPI: create group for synchronizing on tasks

    mtapi_group_attributes_t mtapi_group_attributes;

    mtapi_groupattr_init(
        &mtapi_group_attributes,
        &mtapi_status
    );

    mtapi_groupattr_set(
        &mtapi_group_attributes,
        // ...
    );

    group =  mtapi_group_create(
        MTAPI_GROUP_ID_NONE,
        &mtapi_group_attributes,
        &mtapi_status
    );

    // END MTAPI

    // If fewer than 2 arguments then exit program
    if(argc < 3 - 1) {
        printf("Error: Too few arguments!\n");
        exit(-1);
    }

    // Variables
    int a, b;
    unsigned int wave, waves, i, elements;
    unsigned int np, mp, line, j;

    // Load sequences
    unsigned int k, l;
    FILE *fp;
    fp = fopen(argv[1], "r");
    if(fp == MTAPI_NULL) {
        printf("Error: Could not open file!\n");
        exit(0);
    }
    unsigned int q, n;
    fscanf(fp, "%d %d\n", &q, &n);
    char **sequences = (char**)malloc(q * sizeof(char *));
    for(a = 0; a < q; ++a) {
        sequences[a] = malloc((n + 1) * sizeof(char));
        fscanf(fp, "%s\n", sequences[a]);
    }
    fclose(fp);
    unsigned int m = n;

    // Declare and define open, extension gap penalties
    unsigned int open = atoi(argv[2]);
    unsigned int extension = atoi(argv[3]);

    // Allocate matrices h, e, f
    int **h, **e, **f;
    h = (int**)malloc((n + 1) * sizeof(int *));
    e = (int**)malloc((n + 1) * sizeof(int *));
    f = (int**)malloc((n + 1) * sizeof(int *));
    for(i = 0; i < n + 1; ++i) {
        h[i] = (int*)malloc((m + 1) * sizeof(int));
        e[i] = (int*)malloc((m + 1) * sizeof(int));
        f[i] = (int*)malloc((m + 1) * sizeof(int));
    }

    waves = n + m - 1;
    for(k = 0; k < q; ++k) {
        for(l = k + 1; l < q; ++l) {
            // Write zeroes to top and left corner of matrices h, e, f
            for(i = 0; i < n + 1; ++i) h[i][0] = e[i][0] = f[i][0] = 0;
            for(i = 1; i < m + 1; ++i) h[0][i] = e[0][i] = f[0][i] = 0;
            // Start computing elements along each wave
            for(wave = 0; wave < waves; ++wave) {
                // 0 <= wave < n-1
                if(wave < n-1) {
                    elements = wave+1;
                    np = wave+1;
                    mp = 0+1;
                    line = (n-1)-wave;
                }
                // n-1 <= wave < m
                else if(wave < m) {
                    elements = n;
                    np = n-1+1;
                    mp = wave-(n-1)+1;
                    line = mp;
                }
                // m <= wave < m+n-1
                else {
                    elements = n-1-(wave-m);
                    np = n-1+1;
                    mp = wave-(n-1)+1;
                    line = mp;
                }
                for(i = 0; i < elements; ++i) {
                    // MTAPI: copy arguments and start task
                    // copy arguments
                    element_args args;
                    args.np = np; args.mp = mp; args.i = i;
                    args.k = k; args.l = l;
                    args.h = h; args.e = e; args.f = f;
                    args.sequences = sequences;
                    args.open = open; args.extension = extension;

                    // configure task attributes
                    mtapi_task_attributes_t mtapi_task_attributes;

                    mtapi_taskattr_init(
                        &mtapi_task_attributes,
                        &mtapi_status
                    );

                    mtapi_task_set_attribute(
                        &mtapi_task_attributes,
                        // ...
                    );

                    // start task (remark: should be not so fine grain,
                    // use chunks!)
                    mtapi_task_start(
                        MTAPI_TASK_ID_NONE,
                        job,                   // job
                        &args,                 // arguments struct
                        sizeof(element_args),  // size of arg. struct
                        MTAPI_NULL,                  // result buffer
                        0,                     // size of result buffer
                        &mtapi_task_attributes,
                        group,                 // tasks belong to a group
                        &mtapi_status
                    );
                    // END MTAPI
                }
                // MTAPI: wait for task group completion
                mtapi_group_wait_all(
                    group,          // group handle
                    MTAPI_INFINITE, // no time out
                    &mtapi_status
                );
                // END MTAPI
            }
        }
    }

    // write results (...)

    // finalize MTAPI
    mtapi_finalize(
        &mtapi_status
    );
    return(0);
}
```

5. Appendix A: Acknowledgements

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

Working Group

Dunni Aribuki, University of Houston
Peleg Aviely, Plurality
Eric Biscondi, Texas Instruments
Sunita Chandrasekara, University of Houston
Karol Desnos, Institut National des Sciences Appliquées de Rennes
Daniel Forsgren, Enea
Urs Gleim (chair), Siemens AG
Ted Gribb, Polycore Software, Inc.
Jim Holt, Freescale
Filip Moerman, Texas Instruments
Jeff Niemann, Qualcomm Innovation Center, Inc.
Maxime Pelcat, Institut National des Sciences Appliquées de Rennes
Raymond Richardson, Wind River

Credits

The MTAPI working group also would like to thank all the people who provided input and helped us to improve the specification:

Etem Deniz, Boğaziçi Üniversitesi
Erik Duymelinck, Amdahl Software
Michele Reese, Freescale
Eric Harper, ABB
Thomas Henties, Siemens AG
Ran Katzur, Texas Instruments
Bjoern Knafla, Codeplay Software Ltd.
Anton Leontiev, ELVEES NeoTek CJSC
Kenn Luecke, Boeing
Kevin Martin, Université de Bretagne-Sud
Girish Mundada, Tenasic
Luis Miguel Pinho, Instituto Superior de Engenharia do Porto
Mike Thyer, UltraSoC Technologies Ltd.
George Zaki, University of Maryland

6. Appendix B: Header Files

6.1 mca.h

```
/*
 * mca.h
 * Version 2.000, October 2010
 */

#ifndef MCA_H
#define MCA_H

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

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

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

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

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

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

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

#ifndef MCA_BUF_ALIGN
#define MCA_BUF_ALIGN
#endif

/*
 * MCA organization id's (for assignment of organization specific attribute numbers)
 */
#define MCA_ORG_ID_PSI 0      /* PolyCore Software, Inc. */
#define MCA_ORG_ID_FSL 1      /* Freescale, Inc. */
#define MCA_ORG_ID_MGC 2      /* Mentor Graphics, Corp. */
#define MCA_ORG_ID_TBA 3      /* To be assigned */
/* And so forth */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* MCA_H */
```

6.2 `mtapi.h` Example

```
/**
 * \file
 *
 * \brief MTAPI header contains the public MTAPI API and data type definitions.
 *
 * This file defines the MTAPI API. it has to be included by any application
 * using MTAPI.
 *
 * \copyright
 * Copyright (c) 2013, The Multicore Association.
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 * (1) Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * (2) Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * (3) Neither the name of the Multicore Association nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * \note
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#pragma once
#include "threads.h"  /* C11 threads */


/*****************************************************************************/
/* IMPLEMENTATION SPECIFIC DEFINITIONS */

extern const struct timespec TEN_SECONDS;
extern const struct timespec MTAPI_INFINITE;
extern const struct timespec MTAPI_NOWAIT;

/*****************************************************************************/
/* BASIC DEFINITIONS */

#define MTAPI_IN const                  /**< marks input parameters */
#define MTAPI_OUT                       /**< marks output parameters */
#define MTAPI_INOUT                     /**< marks in/out parameters */

#define MTAPI_QUEUE_DEFAULT_SIZE 1024   /**< default size for MTAPI queues
                                             (implementation specific) */


/*****************************************************************************/
/* BASIC DATA TYPES */

/* the MTAPI data types */
typedef mca_int_t       mtapi_int_t;
typedef mca_int8_t      mtapi_int8_t;
typedef mca_int16_t     mtapi_int16_t;
typedef mca_int32_t     mtapi_int32_t;
typedef mca_int64_t     mtapi_int64_t;

typedef mca_uint_t      mtapi_uint_t;
typedef mca_uint8_t     mtapi_uint8_t;
typedef mca_uint16_t    mtapi_uint16_t;
typedef mca_uint32_t    mtapi_uint32_t;
typedef mca_uint64_t    mtapi_uint64_t;

typedef mca_domain_t    mtapi_domain_t;
typedef mca_node_t      mtapi_node_t;
typedef mca_timeout_t   mtapi_timeout_t;

typedef mca_boolean_t mtapi_boolean_t;
typedef mtapi_uint_t  mtapi_size_t;


struct mtapi_info_struct{
    mtapi_int_t lenth;    /* length in byte */
    void* data;
};
typedef struct mtapi_info_struct mtapi_info_t;


typedef unsigned long * mtapi_affinity_t;

#define MTAPI_NULL    0

/*******************************************************************************/
/* BASIC enumerations */

enum mtapi_status_enum {
    /* generic */
    MTAPI_SUCCESS,                      /**< sucess, no error */
    MTAPI_TIMEOUT,                      /**< timeout was reached */
    MTAPI_ERR_PARAMETER,                /**< invalid parameter */
    MTAPI_ERR_ATTR_READONLY,            /**< tried to write a read-only attribute */
    MTAPI_ERR_ATTR_NUM,                 /**< invalid attribute number */
    MTAPI_ERR_ATTR_SIZE,                /**< invalid attribute size */

    /* node specific */
    MCAPI_ERR_NODE_INITFAILED,          /**< general error in node initialization */
    MTAPI_ERR_NODE_INITIALIZED,
      /**< \a mtapi_initialize called for a node that  already had been initialized */
    MTAPI_ERR_NODE_INVALID,             /**< The node id is not valid */
    MTAPI_ERR_DOMAIN_INVALID,           /**< the domain id is not valid */
    MTAPI_ERR_NODE_NOTINIT,             /**< the node is not initialized */

    /* action specific */
    MTAPI_ERR_ACTION_INVALID,
      /**< The action id is not a valid action id, i.e., no action was crated for that
           ID or the action has been deleted. */
    MTAPI_ERR_ACTION_EXISTS,
      /**< mtapi_action_create called with an ID of an action that already had been created */
    MTAPI_ERR_ACTION_LIMIT,
      /**< exceeded maximum number of actions allowed */
    MTAPI_ERR_ACTION_NUM_INVALID,
      /**< The numeber of actions passed to mtapi_task_start or mtapi_queue_create is lower
           than 1. */
    MTAPI_ERR_ACTION_FAILED,
      /**< status that can be passed to the runtime by \a mtapi_context_status_set if the task
           could not be completed as intended */
    MTAPI_ERR_ACTION_CANCELLED,
      /**< status that can be passed to the runtime by \a mtapi_context_status_set if if the
           task execution is cancelled */
    MTAPI_ERR_ACTION_DELETED,
      /**< All actions associated with the task have been deleted before the execution of the
           task was started or the error code has been set in the action code to
           MTAPI_ERR_ACTION_DELETED by \a mtapi_action_result_set. */
    MTAPI_ERR_ACTION_DISABLED,
      /**< All actions associated with the task have been disabled before the execution of the
           task was started or the error code has been set in the action code to
           MTAPI_ERR_ACTION_DISABLED by \a mtapi_action_result_set. */
    MTAPI_ERR_CONTEXT_OUTOFCONTEXT,
      /**< returned if action code is not called in the context of a task execution. This
           function must be used in an action function only. The action function must be called
           from the MTAPI runtime system */

    /* action specific */
    MTAPI_ERR_JOB_INVALID,               /**< invalid job handle or job ID */

    /* queue specific */
    MTAPI_ERR_QUEUE_INVALID,            /**< argument is not a valid queue handle or ID */
    MTAPI_ERR_QUEUE_DELETED,            /**< a queue that no longer exists is passed as an
                                             argument */
    MTAPI_ERR_QUEUE_DISABLED,            /**< a queue that no longer exists is passed as an
                                             argument */

    /* group specific */
    MTAPI_GROUP_COMPLETED,
      /**< group completed, i.e., there are no more task to wait for in the group when
           waiting with \a mtapi_group_wait_any */

    /* others */
    MTAPI_ERR_UNKNOWN,                  /**< unknown error */
    MTAPI_ERR_BUFFER_SIZE,              /**< buffer size mismatch */
    MTAPI_ERR_RESULT_SIZE,              /**< result buffer size mismatch (e.g., in \a
                                             mtapi_task_wait) */
    MTAPI_ERR_ARG_SIZE,                 /**< invalid argument size */
    MTAPI_ERR_WAIT_PENDING,             /**< mtapi_*_wait called twice on a group or task
                                             which has not finished */


    /* unsupported functions */
    MTAPI_ERR_FUNC_NOT_IMPLEMENTED,     /**< The MTAPI function called is not implemented by
the runtime system. */
    MTAPI_ERR_ARG_NOT_IMPLEMENTED,
      /**< The MTAPI function called is implemented by the runtime, but it does not
           support the agruments passed. */

    /* features that may be not supported by some implementations */
    MTAPI_ERR_RUNTIME_REMOTETASKS_NOTSUPPORTED,
      /**< The Runtime system does not support remote tasks. This allows lighter
           implementations for shared memory environments. */
    MTAPI_ERR_RUNTIME_LOADBALANCING_NOTSUPPORTED,
      /**< This error is returns when more than one action is passed to mtapi_task_start or
           mtapi_queue_create and if the runtime system does not implement load balancing
           between nodes. This allows light MTAPI implementation for systems not having the
           requirment for inter-node laod-balancing */
};
typedef enum mtapi_status_enum mtapi_status_t; /**< defines the MTAPI error codes */

enum mtapi_task_state_enum {
    MTAPI_TASK_CREATED,
    MTAPI_TASK_SCHEDULED,
    MTAPI_TASK_RUNNING,
    MTAPI_TASK_WAITING,
    MTAPI_TASK_DELETED,
    MTAPI_TASK_CANCELLED,               /**< \a MTAPI_TASK_CANCELLED is the only value
           specified by the MTAPI specification, the other are implementation specific and can
           be used for debugging purposes. */
    MTAPI_TASK_COMPLETED
};
typedef enum mtapi_task_state_enum mtapi_task_state_t; /**< internal task state */


enum mtapi_notification_enum {
    MTAPI_NOTIF_PREFETCH,               /**< implementation specific example */
    MTAPI_NOTIF_EXECUTE_NEXT            /**< implementation specific example */
};
typedef enum mtapi_notification_enum mtapi_notification_t; /**< runtime notification */


/** node attributes, to be extended for implementation specific attribtes */
enum mtapi_node_atttributes_enum {
    MTAPI_NODES_NUMCORES                /**< number of cores provided by the node */
};
/** size of the \a MTAPI_NODES_NUMCORES attribute */ /* = 0 since it is not a pointer */
#define MTAPI_NODES_NUMCORES_SIZE 0

/** task attributes */
enum mtapi_task_atttributes_enum {
    MTAPI_TASK_DETACHED,
      /**< task is detached, i.e., the runtime system cared about deleting internal data
```

```
           strauctures representing the the task; detached tasks cannot be accessed via task
handles */
    MTAPI_TASK_INSTANCES
      /**< indicates how many parallel instances of task shall be started by MTAPI; the default
           case is that each task is executed exactly once, setting this value to \a n, the
           corresponding action code will be executed n times, if possible in parallel */
};
/** size of the \a MTAPI_TASK_DETACHED attribute */ /* = 0 since it is not a pointer */
#define MTAPI_TASK_DETACHED_SIZE 0
/** size of the \a MTAPI_TASK_INSTANCES attribute */ /* = 0 since it is not a pointer */
#define MTAPI_TASK_INSTANCES_SIZE 0



/** action attributes */
enum mtapi_action_atttributes_enum {
    MTAPI_ACTION_AFFINITY
};
/** size of the \a MTAPI_ACTION_AFFINITY_MASK attribute */
#define MTAPI_ACTION_AFFINITY_MASK_SIZE sizeof(mtapi_affinity_t)


/*******************************************************************************/
/* ATTRIBUTES AND DEFAULT INITIALIZERS */

/** node attribtes, to be adapted implemenation speciifically */
struct mtapi_node_attributes_struct{
    mtapi_int_t some_value;
};

/* needed for definition of MTAPI_DEFAULT_NODE_ATTRIBUTES, see below */
#define _MTAPI_NODE_ATTRIBUTES_INITIALIZER {1}

struct mtapi_action_attributes_struct{
    mtapi_int_t some_value;
};

/* needed for definition of MTAPI_DEFAULT_ACTION_ATTRIBUTES, see below */
#define _MTAPI_ACTION_ATTRIBUTES_INITIALIZER {1}

struct mtapi_task_attributes_struct{
    mtapi_int_t num_instances;
};

/* needed for definition of MTAPI_DEFAULT_TASK_ATTRIBUTES, see below */
#define _MTAPI_TASK_ATTRIBUTES_INITIALIZER {1}

struct mtapi_queue_attributes_struct{
    mtapi_int_t some_value;
};

/* needed for definition of MTAPI_DEFAULT_QUEUE_ATTRIBUTES, see below */
#define _MTAPI_QUEUE_ATTRIBUTES_INITIALIZER {1}

struct mtapi_group_attributes_struct{
    mtapi_int_t some_value;
};

/* needed for definition of MTAPI_DEFAULT_GROUP_ATTRIBUTES, see below */
#define _MTAPI_GROUP_ATTRIBUTES_INITIALIZER {1}

typedef struct mtapi_node_attributes_struct mtapi_node_attributes_t;
typedef struct mtapi_action_attributes_struct mtapi_action_attributes_t;
typedef struct mtapi_task_attributes_struct mtapi_task_attributes_t;
typedef struct mtapi_queue_attributes_struct mtapi_queue_attributes_t;
typedef struct mtapi_group_attributes_struct mtapi_group_attributes_t;

extern mtapi_node_attributes_t _mtapi_default_node_attributes;

/** short form for using the default node attributes */
#define MTAPI_DEFAULT_NODE_ATTRIBUTES &_mtapi_default_node_attributes

extern mtapi_action_attributes_t _mtapi_default_action_attributes;

/** short form for using the default action attributes */
```

```
#define MTAPI_DEFAULT_ACTION_ATTRIBUTES &_mtapi_default_action_attributes

extern mtapi_task_attributes_t _mtapi_default_task_attributes;

/** short form for using the default task attributes */
#define MTAPI_DEFAULT_TASK_ATTRIBUTES &_mtapi_default_task_attributes

extern mtapi_queue_attributes_t _mtapi_default_queue_attributes;

/** short form for using the default queue attributes */
#define MTAPI_DEFAULT_QUEUE_ATTRIBUTES &_mtapi_default_queue_attributes

extern mtapi_group_attributes_t _mtapi_default_group_attributes;

/** short form for using the default group attributes */
#define MTAPI_DEFAULT_GROUP_ATTRIBUTES &_mtapi_default_group_attributes


/******************************************************************************/
/* FUNCTION TYPES */

typedef void (*mtapi_action_function_t)(
    void* args,                         /* arguments */
    mtapi_size_t args_size,             /* length of arguments */
    void* result_buffer,                /* buffer for storing results */
    mtapi_size_t result_buffer_size,    /* length of result_buffer */
    void* node_local_data,              /* node-local data,
                                           shared data by several
                                           tasks exeecuted on the
                                           same node */
    mtapi_size_t node_local_data_size,  /* length of shared data */
    mtapi_task_context_t * context      /* MTAPI task context provided
                                           by the runtime systems
                                           identifying the current task
                                           for calling back the runtime
                                           system from the action function */
);

/******************************************************************************/
/* HANDLES and IDs*/

typedef mtapi_uint_t mtapi_job_id_t;
typedef mtapi_uint_t mtapi_task_id_t;
typedef mtapi_uint_t mtapi_group_id_t;

#define MTAPI_TASK_ID_NONE 0
#define MTAPI_GROUP_ID_NONE 0
#define MTAPI_QUEUE_ID_NONE 0
#define MTAPI_ACTION_ID_NONE 0


/* local action actiom */
struct mtapi_action_hndl_struct;
typedef struct mtapi_action_hndl_struct  mtapi_action_hndl_t;


/* remote actiom */
struct _mtapi_action_remote_struct;
typedef struct _mtapi_action_remote_struct  _mtapi_action_remote_t;


/* local or remote action */
struct _mtapi_action_ref_struct;
typedef struct _mtapi_action_ref_struct  _mtapi_action_ref_t;

/* job TODO: should be defined as pointer to internal struct in array (_matpi_job_list) */
struct mtapi_job_hndl_struct;
typedef struct mtapi_job_hndl_struct  mtapi_job_hndl_t;


typedef mtapi_int_t mtapi_queue_id_t;

struct mtapi_queue_hndl_struct;
typedef struct mtapi_queue_hndl_struct  mtapi_queue_hndl_t;
```

```
struct mtapi_group_struct;

struct mtapi_task_hndl_struct;

typedef struct mtapi_task_hndl_struct mtapi_task_hndl_t;


/******************************************************************************/
/* BASIC CONSTANTS */

#define MTAPI_TRUE  1
#define MTAPI_FALSE 0


/******************************************************************************/
/* SYNTACTIC SUGAR */

#define MTAPI_GROUP_NONE 0


/******************************************************************************/
/******************************************************************************/
/* RUNTIME INIT & SHUTDOWN */

void mtapi_initialize(
    MTAPI_IN mtapi_domain_t domain_id,
    MTAPI_IN mtapi_node_t node_id,
    MTAPI_IN mtapi_node_attributes_t* attributes,
    MTAPI_OUT mtapi_info_t* mtapi_info,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_finalize(
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_nodeattr_init(
    MTAPI_OUT mtapi_node_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_nodeattr_set(
    MTAPI_INOUT mtapi_node_attributes_t* attributes,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* CORE AFFINITY MASKS */


/* initial affinity mask is dependet on 'affinity', setting affinity to
   MTAPI_FALSE similar to POSIX CPU_ZERO */
void mtapi_affinity_init (
    MTAPI_OUT mtapi_affinity_t* mask,
    MTAPI_IN mtapi_boolean_t affinity,
    MTAPI_OUT mtapi_status_t* status
);


/* similar to POSIX CPU_SET and CPU_CLR */
void mtapi_affinity_set (
    MTAPI_INOUT mtapi_affinity_t* mask,
    MTAPI_IN mtapi_uint_t core_num,
    MTAPI_IN mtapi_boolean_t affinity,
    MTAPI_OUT mtapi_status_t* status
);

/* similar to POSIX CPU_ISSET */
mtapi_boolean_t mtapi_affinity_get (
    MTAPI_OUT mtapi_affinity_t* mask,
    MTAPI_IN mtapi_uint_t core_num,
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* ACTIONS */

void mtapi_actionattr_init(
    MTAPI_OUT mtapi_action_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_actionattr_set(
    MTAPI_INOUT mtapi_action_attributes_t* attributes,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);

mtapi_action_hndl_t mtapi_action_create(
    MTAPI_IN mtapi_job_id_t job_id,
    MTAPI_IN mtapi_action_function_t function,
    MTAPI_IN void* node_local_data,                    /* shared data by several tasks */
    MTAPI_IN mtapi_size_t node_local_data_size,        /* length of shared data */
    MTAPI_IN mtapi_action_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_action_set_attribute (
    MTAPI_IN mtapi_action_hndl_t action,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_action_get_attribute (
    MTAPI_IN mtapi_action_hndl_t action,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_OUT void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);


void mtapi_action_delete(
    MTAPI_IN mtapi_action_hndl_t action,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_action_disable(
    MTAPI_IN mtapi_action_hndl_t action,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_action_enable(
    MTAPI_IN mtapi_action_hndl_t action,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_context_status_set(
    MTAPI_INOUT mtapi_task_context_t* task_context,
    MTAPI_IN mtapi_status_t error_code,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_context_runtime_notify (
    MTAPI_IN mtapi_task_context_t* task_context,
    MTAPI_IN mtapi_notification_t notification,
        MTAPI_IN void* data,
        MTAPI_IN mtapi_size_t data_size,
        MTAPI_OUT mtapi_status_t* status
```

```
);


mtapi_task_state_t mtapi_context_taskstate_get(
    mtapi_task_context_t* task_context,
    MTAPI_OUT mtapi_status_t* status
);

/* derived from OpenMP's omp_get_thread_num(); */
mtapi_uint_t mtapi_context_instnum_get(
    mtapi_task_context_t* task_context,
    MTAPI_OUT mtapi_status_t* status
);

/* derived from OpenMP's omp_get_num_threads(); */
mtapi_uint_t mtapi_context_numinst_get(
    mtapi_task_context_t* task_context,
    MTAPI_OUT mtapi_status_t* status
);

/* get the current core - for debugging purposes */
mtapi_uint_t mtapi_context_corenum_get (
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* JOBS */

mtapi_job_hndl_t mtapi_job_get(
    MTAPI_IN mtapi_job_id_t job_id,
    MTAPI_IN mtapi_domain_t domain_id,
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* QUEUES */

void mtapi_queueattr_init (
    MTAPI_OUT mtapi_queue_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_queueattr_set (
    MTAPI_INOUT mtapi_queue_attributes_t* attributes,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);

mtapi_queue_hndl_t mtapi_queue_create (
    MTAPI_IN mtapi_queue_id_t queue_id,
        MTAPI_IN mtapi_job_hndl_t job,
    MTAPI_IN mtapi_queue_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_queue_set_attribute (
        MTAPI_IN mtapi_queue_hndl_t queue,
        MTAPI_IN mtapi_uint_t attribute_num,
        MTAPI_IN void* attribute,
        MTAPI_IN mtapi_size_t attribute_size,
        MTAPI_OUT mtapi_status_t* status
);


mtapi_queue_hndl_t mtapi_queue_get(
    MTAPI_IN mtapi_queue_id_t queue_id,
    mtapi_domain_t domain_id,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_queue_delete(
    MTAPI_IN mtapi_queue_hndl_t queue,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_queue_disable(
    MTAPI_IN mtapi_queue_hndl_t queue,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_queue_enable(
    MTAPI_IN mtapi_queue_hndl_t queue,
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* TASK GROUPS */

void mtapi_groupattr_init (
    MTAPI_OUT mtapi_group_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_groupattr_set (
    MTAPI_INOUT mtapi_group_attributes_t* attributes,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);

mtapi_group_hndl_t mtapi_group_create (
    MTAPI_IN mtapi_group_id_t group_id,
    MTAPI_IN mtapi_group_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_group_set_attribute (
        MTAPI_IN mtapi_group_hndl_t group,
        MTAPI_IN mtapi_uint_t attribute_num,
        MTAPI_OUT void* attribute,
        MTAPI_IN mtapi_size_t attribute_size,
        MTAPI_OUT mtapi_status_t* status
);

void mtapi_group_wait_all (
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_group_wait_any(
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_OUT void** result,
    MTAPI_IN mtapi_timeout_t timeout,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_group_delete(
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_OUT mtapi_status_t* status
);


/******************************************************************************/
/* TASKS */

void mtapi_taskattr_init (
    MTAPI_OUT mtapi_task_attributes_t* attributes,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_taskattr_set (
    MTAPI_INOUT mtapi_task_attributes_t* attributes,
    MTAPI_IN mtapi_uint_t attribute_num,
    MTAPI_IN void* attribute,
    MTAPI_IN mtapi_size_t attribute_size,
    MTAPI_OUT mtapi_status_t* status
);

mtapi_task_hndl_t mtapi_task_start (
    MTAPI_IN mtapi_task_id_t task_id,
    MTAPI_IN mtapi_job_hndl_t job,
    MTAPI_IN void* arguments,
    MTAPI_IN mtapi_size_t arguments_size,
    MTAPI_OUT void* result_buffer,       /* pointer to result buffer */
    MTAPI_IN mtapi_size_t result_size,   /* size of one result */
    MTAPI_IN mtapi_task_attributes_t* attributes,
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_OUT mtapi_status_t* status
);

mtapi_task_hndl_t mtapi_task_enqueue (
    MTAPI_IN mtapi_task_id_t task_id,
    MTAPI_IN mtapi_queue_hndl_t queue,
    MTAPI_IN void* arguments,
    MTAPI_IN mtapi_size_t arguments_size,
    MTAPI_OUT void* result_buffer,       /* pointer to result buffer */
    MTAPI_IN mtapi_size_t result_size,   /* size of one result */
    MTAPI_IN mtapi_task_attributes_t* attributes,
    MTAPI_IN mtapi_group_hndl_t group,
    MTAPI_OUT mtapi_status_t* status
);

void mtapi_task_wait(
    MTAPI_IN mtapi_task_hndl_t task,    /* task handle */
    MTAPI_IN mtapi_timeout_t timeout,   /* time out */
    MTAPI_OUT mtapi_status_t* status    /* status */
);

void mtapi_task_cancel(
    MTAPI_IN mtapi_task_hndl_t task,
    MTAPI_OUT mtapi_status_t* status
);
```

7. Appendix C: MTAPI License Agreement

1. General. The specification and/or documentation accompanying this License whether on disk, in read only memory, on any other media or in any other form (collectively the "MTAPI SPECIFICATION") are licensed, not sold, to you by the MULTICORE ASSOCIATION for use only under the terms of this License, and MULTICORE ASSOCIATION reserves all rights not expressly granted to you. The rights granted herein are limited in scope per the terms of this License AND are also limited as to the source of the rights being granted. The rights granted under this License are limited to those rights that MULTICORE ASSOCIATION's members and its licensors' have granted to MULTICORE ASSOCIATION for incorporation and sublicensing as a part of the MULTICORE ASSOCIATION MTAPI SPECIFICATION, and do not include any other rights. The terms of this License will govern any upgrades provided by MULTICORE ASSOCIATION that replace and/or supplement the original MULTICORE ASSOCIATION MTAPI SPECIFICATION, unless such upgrade is accompanied by a separate license in which case the terms of that license will govern.

2. Permitted License Uses and Restrictions. This License allows you to download and use the MULTICORE ASSOCIATION MTAPI SPECIFICATION only as expressly permitted under this License. You may make only that number of copies of the MTAPI SPECIFICATION as are reasonably necessary in order to effectuate the purposes for which the MTAPI SPECIFICATION is intended. You may not make the MTAPI SPECIFICATION available over a network where it could be used by multiple platforms or multiple users at the same time or for use via data hosting, time sharing or service bureau usage. Except as and only to the extent expressly permitted in this License or by applicable law, you may not copy, modify, or create derivative works of the MTAPI SPECIFICATION or any part thereof. THE MTAPI SPECIFICATION IS NOT INTENDED FOR USE IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL SYSTEMS, LIFE SUPPORT MACHINES OR OTHER EQUIPMENT IN WHICH THE FAILURE OF THE MTAPI SPECIFICATION COULD LEAD TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE.

3. Enhancements by MULTICORE ASSOCIATION. In the event that at any time MULTICORE ASSOCIATION makes any enhancement, update, or modification to the MTAPI SPECIFICATION or becomes the owner of any new enhancement, update, or modification to the MTAPI SPECIFICATION, then upon notice to you, you shall have the same right and license to use and exploit the same as it is granted hereunder with respect to the original MTAPI SPECIFICATION. You agree that all rights in and to enhancements, updates, and modifications effected by MULTICORE ASSOCIATION, if any, to the MTAPI SPECIFICATION, shall remain the sole and exclusive property of MULTICORE ASSOCIATION. Nothing contained herein shall obligate MULTICORE ASSOCIATION to create any new enhancement, update or modifications to the MULTICORE ASSOCIATION MTAPI SPECIFICATION, to make such enhancement, update or modifications available free of charge or to provide any maintenance and technical support services.

4. Enhancements by You. You shall have no right to independently modify, improve, or enhance the MULTICORE ASSOCIATION MTAPI SPECIFICATION.

5. Use of Trademark. You agree that you will not, without the prior written consent of MULTICORE ASSOCIATION, use in advertising, publicity, packaging, labeling, or otherwise any trade name, trademark, service mark, symbol, or any other identification owned by MULTICORE ASSOCIATION to identify any of its products or services.

6. Delivery. The MULTICORE ASSOCIATION MTAPI SPECIFICATION will be available for downloading on MULTICORE ASSOCIATION's website in accordance with the current policies and procedures of MULTICORE ASSOCIATION. MULTICORE ASSOCIATION reserves the right to modify such procedures in its sole and absolute discretion.

7. Term and Termination.

a. Term. The term of this Agreement shall continue so long as you are not in Default as set forth in Section 8(b).

b. Termination. MULTICORE ASSOCIATION shall have the right to terminate this Agreement and the License granted herein if you commit an act of or are subject to any Default. A Default means you breach a term or condition of this Agreement, including, but not limited to, a breach of the confidentiality provisions or payment provisions hereof or you assign or purport to assign any of the rights granted herein without the prior written approval of MULTICORE ASSOCIATION. Upon the occurrence of a Default, this Agreement shall immediately terminate. MULTICORE ASSOCIATION's rights as set forth in this Section 8(b) are cumulative and, except as provided herein, are in addition to any other rights MULTICORE ASSOCIATION may have at law or in equity.

c. Effect of Termination. Upon the expiration or termination of this Agreement, the rights granted to you hereunder shall immediately cease and discontinue, and you shall be required to immediately return any and all materials and deliverables provided to you under this Agreement, including without limitation, the MULTICORE ASSOCIATION MTAPI SPECIFICATION. The provisions contained in Sections 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17 shall survive any such termination or expiration.

8. No Warranty. MULTICORE ASSOCIATION provides no warranty for the MULTICORE ASSOCIATION MTAPI SPECIFICATION.

9. Disclaimer of Warranties. You expressly acknowledge and agree that use of the MULTICORE ASSOCIATION MTAPI SPECIFICATION is at your sole risk and that the entire risk as to satisfactory quality, performance, accuracy and effort is with you. The MULTICORE ASSOCIATION MTAPI SPECIFICATION is provided "as is", with all faults and without warranty of any kind, and MULTICORE ASSOCIATION and MULTICORE ASSOCIATION's licensors (collectively referred to as "MULTICORE ASSOCIATION" for the purposes of sections 10 and 11) hereby disclaim all warranties and conditions with respect to the MULTICORE ASSOCIATION MTAPI SPECIFICATION, either express, implied or statutory, including, but not limited to, the implied warranties and/or conditions of merchantability, of satisfactory quality, of fitness for a particular purpose, of accuracy, of quiet enjoyment, and non-infringement of third party rights. MULTICORE ASSOCIATION does not warrant against interference with your enjoyment of the MULTICORE ASSOCIATION MTAPI SPECIFICATION, that the functions contained in the MULTICORE ASSOCIATION MTAPI SPECIFICATION will meet your requirements, that the operation of the MULTICORE ASSOCIATION MTAPI SPECIFICATION will be uninterrupted or error-free, or that defects in the MULTICORE ASSOCIATION MTAPI SPECIFICATION will be corrected. No oral or written information or advice given by MULTICORE ASSOCIATION or an MULTICORE ASSOCIATION authorized representative shall create a warranty. Should the MULTICORE ASSOCIATION MTAPI SPECIFICATION prove defective, you assume the entire cost of all necessary servicing, repair or correction. Some jurisdictions do not allow the exclusion of implied warranties or limitations on applicable statutory rights of a consumer, so the above exclusion and limitations may not apply to you.

10. Potential Misuse of MULTICORE ASSOCIATION MTAPI SPECIFICATION. You hereby acknowledge and represent that you have been expressly warned by MULTICORE ASSOCIATION that the MULTICORE ASSOCIATION MTAPI SPECIFICATION may be incompatible with any application or end-user product, and that such misuse of the MULTICORE ASSOCIATION MTAPI SPECIFICATION could result in significant property damage and/or bodily harm.

11. Limitation of Liability. TO the extent not prohibited by law, in no event shall MULTICORE ASSOCIATION be liable for personal injury, or any incidental, special, indirect or consequential damages whatsoever, including, without limitation, damages for loss of profits, loss of data, business interruption or any other commercial damages or losses, arising out of or related to your use or inability to use the MULTICORE ASSOCIATION MTAPI SPECIFICATION, however caused, regardless of the theory of liability (contract, tort or otherwise) and even if MULTICORE ASSOCIATION has been advised of the possibility of such damages. some jurisdictions do not allow the limitation of liability for personal

injury, or of incidental or consequential damages, so this limitation may not apply to you. In no event shall MULTICORE ASSOCIATION's total liability to you for all damages (other than as may be required by applicable law in cases involving personal injury) exceed the amount of fifty U.S. dollars ($50.00). The foregoing limitations will apply even if the above stated remedy fails of its essential purpose.

12. Export Law Assurances. You may not use or otherwise export or re-export the MULTICORE ASSOCIATION MTAPI SPECIFICATION except as authorized by United States law and the laws of the jurisdiction in which the MULTICORE ASSOCIATION MTAPI SPECIFICATION was obtained. In particular, but without limitation, the MULTICORE ASSOCIATION MTAPI SPECIFICATION may not be exported or re-exported (a) into (or to a national or resident of) any U.S. embargoed countries (currently Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria), or (b) to anyone on the U.S. Treasury Department's list of Specially Designated Nationals or the U.S. Department of Commerce Denied Person's List or Entity List. By using the MULTICORE ASSOCIATION MTAPI SPECIFICATION, you represent and warrant that you are not located in, under control of, or a national or resident of any such country or on any such list.

13. Relationship of Parties. Neither this Agreement, nor any terms and conditions contained herein, may be construed as creating or constituting a partnership, joint venture, or agency relationship between the parties. Neither party will have the power to bind the other or incur obligations on the other party's behalf without the other party's prior written consent.

14. Waiver. No failure of either party to exercise or enforce any of its rights under this Agreement will act as a waiver of such rights.

15. Controlling Law and Severability. This License will be governed by and construed in accordance with the laws of the State of California, as applied to agreements entered into and to be performed entirely within California between California residents. This License shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly excluded. If for any reason a court of competent jurisdiction finds any provision, or portion thereof, to be unenforceable, the remainder of this License shall continue in full force and effect.

16. Complete Agreement; Governing Language. This License constitutes the entire agreement between the parties with respect to the use of the MULTICORE ASSOCIATION MTAPI SPECIFICATION licensed hereunder and supersedes all prior or contemporaneous understandings regarding such subject matter. No amendment to or modification of this License will be binding unless in writing and signed by MULTICORE ASSOCIATION. Any translation of this License is done for local requirements and in the event of a dispute between the English and any non-English versions, the English version of this License shall govern.

8. Appendix D: References

[1] Ayguadé et al., The Design of OpenMP Tasks, IEEE TPDS March 2009

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall und Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In: Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207–216. ACM, 1995.

[3] R. D. Blumofe und C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In: Symposium on Foundations of Computer Science (FOCS), pages 356–368. IEEE, 1994.

[4] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory, Parallel Programming. MIT Press, 2008.

[5] Intel Corporation. Intel Threading Building Blocks: Reference Manual, 2011. Document No. 315415-015US.

[6] Intel Corporation. Intel Threading Building Blocks: Tutorial, 2008.

[7] International Standard ISO/IEC 9899:2011, Programming languages – C

[8] LLC QuickThread Programming. Quickthread threading toolkit programmer's reference manual v 1.0.1. http://www.quickthreadprogramming.com/, 2009. [Online; accessed 15 January 2012].

[9] OpenMP Architecture Review Board. OpenMP Application Program Interface (version 3.1), 2011.

[10] J. Reinders. Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly, 2007

[11] Aki Seppänen and Tommi Mikkonen. Porting legacy applications to multicore: Experiences from an industrial system. In PDP '09: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pages 127–132, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Supercomputing Technologies Group, MIT Laboratory for Computer Science. Cilk 5.4.6 Reference Manual.

[13] Wikipedia. Cilk — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cilk&oldid=463493244, 2011. [Online; accessed 15 February 2012].