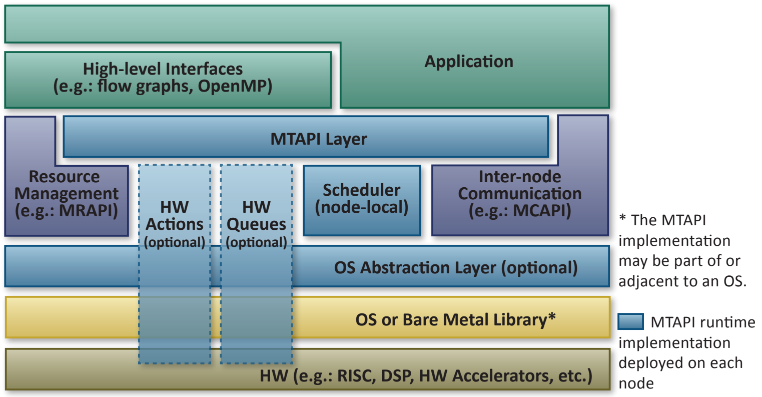
Multicore Task Management API (MTAPI™) In a Nutshell

MTAPI provides an API which allows parallel embedded software to be designed by abstracting the hardware details and letting the software developer focus on the parallel solution of the problem.

MTAPI Overview

MTAPI™ defines a C-language API for creating low-level lightweight tasks that can delegate their execution to hardware or software nodes being part of a system consisting of homogeneous or heterogeneous processors and/or processor cores. The granularity of tasks is intended to be fine (systems should be able to manage thousands of tasks). The underlying MTAPI runtime system can be based on an OS, run in a bare metal environment, or on a hypervisor (see MTAPI reference architecture on the right). The runtime system manages tasks for the whole heterogeneous system. MTAPI works together with current and upcoming Multicore Association® standards.



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

MTAPI Terms

The MTAPI abstractions are aligned with MCAPI and MRAPI. A system is logically decomposed hierarchically into domains, nodes and cores. Tasks execute a job implemented by one more actions. The scheduling policy can be controlled by task and action attributes or by using dedicated queues associated with a job. The following alphabetic list describes the most important terms in more detail:

Action (*mtapi_action_hndl_t*): Every action implements a job. Implementation of a job may be hardware or software-defined. An action must be created (*mtapi_action_create*) on the node that implements it. Tasks can access actions only indirectly via the job identifier (*mtapi_job_get*) on the same or another node. A hardware-implemented action does not need to be created. It is possible to start sub-tasks from the action code.

Action function (*mtapi_action_function_t*): Function with a certain signature containing the action code for software-defined actions.

Affinity (*mtapi_affinity_t*): Mapping of actions to processor cores for execution by a core affinity mask (implementation-specific).

Core: Undividable processing element. Two cores can share resources, for instance memory or ALUs for hyper-threaded cores. The core notion is necessary for core affinity, but implementation-specific.

Domain (*mtapi_domain_t*): Comparable to a subnet in a network or a namespace for unique names and identifiers (e.g., node, actions, queues). A domain contains a set of nodes. Actions and queues may or may not be shared across domains. The MTAPI_DOMAIN_SHARED attribute set to MTAPI_FALSE disables cross-domain actions and queues.

Handle: A handle is a local reference to an entity (e.g.: job, task, queue, group) that cannot be used by another node other than the one that requested it. Handles referencing remote entities are created by *mtapi_queue_get* and *mtapi_job_get*.

Identifier (*mtapi_job_id_t* and *mtapi_queue_id_t*): A unique integer identifier referencing an entity like a job or a queue. The ID is necessary to retrieve a handle on an entity that was possibly defined on another node. If a queue is created with no valid ID (MTAPI_QUEUE_ID_NONE), it can only be used on the node where it was created (referenced by node-local *handle*). Tasks and groups have identifiers (*mtapi_task_id_t* and *mtapi_group_id_t*) are for debugging purposes only and are not referenced from other nodes.

Job: A piece of processing implemented by an action. Several actions can implement the same job based on different hardware resources (for instance a job can be implemented on a DSP and a general purpose core, or a job can be both implemented in hardware and in software). Each job has a unique identifier.

Multi-instance task: Starting a multi-instance task, triggers execution of actions implementation a given job MTAPI_TASK_INSTANCES times, where MTAPI_TASK_INSTANCES is a task attribute. The corresponding action code can retrieve the currently executed instance (*mtapi_context_instnum_get*) via the task context.

Node (*mtapi_node_t*): Processing element that can have some self-managed parallelism. A node can be a process, thread, thread pool, instance of an operating system, hardware accelerator, or multicore processor. A node can represent a subset of cores in a given architecture.

Queue (*mtapi_queue_hndl_t*): A software or hardware entity in which tasks are enqueued in a given order. The queue can then either ensure in-order execution of tasks or manage parallel execution of tasks (load balancing) onto parallel nodes. The application, in the latter case, can enqueue and forget tasks. A queue is either associated with job that is implemented by one more actions action. A queue is created (*mtapi_queue_create*) on a given node and can be retrieved (*mtapi_queue_get*) on another node, from its identifier. Queues may offer implementation specific scheduling policies.

Runtime information (*mtapi_info_t*): Information on the runtime system implementation returned by mtapi_initialize. MTAPI-specified information contains implementation_version, mtapi_version, organization_id, number_of_domains, and number_of_nodes. Implementation-specific information must be documented by the implementer.

Status (*mtapi_status_t*): Contains success or error code of the last API function call.

Task (*mtapi_task_hndl_t*): One execution of a job resulting in the invocation of an action implementing the job associated with some data to be processed. It can run only once and is run-to-completion. It is possible to wait for completion of a task (*mtapi_task_wait*) on the node that invoked it. A task is intended to be executed in parallel to the code that started it. A task handle is local to a node. It cannot be accessed from another node. A task can be cancelled (*mtapi_task_cancel*). Cancellation order is transmitted to the action code via the task context (see below). If a task is detached at creation, MTAPI runtime system will automatically delete it after completion. Neither waiting nor cancellation is possible for detached tasks.

(continued >)

MTAPI Terms (continued)

Task context (*mtapi_task_context_t*): Information about the task, available in the corresponding action.

Task group (*mtapi_group_hndl_t*): A set of tasks (detached or not) that can be waited on for completion (*mtapi_group_wait_all*, *mtapi_group_wait_any*) locally to the node that created the group and belonging tasks.

MTAPI Use Case Scenarios

In the following use case scenarios, a node Y is responsible for scheduling tasks. Nodes X, Y, and Z (every node is a physical processor, for example) run an implementation of the MTAPI runtime system.

Scenario 1 - Any Task: In this scenario, synchronization and timing of tasks is done explicitly by the application running on Y. The jobs are executed by X where the action is created:

1.1. Create the action object A on a node X and assign a job identifier (*mtapi_action_create*, if A is software-implemented – hardware-implemented actions do not need to be created). Obtain an job handle of A on node Y (*mtapi_job_get*). If Y=X, *mtapi_action_create* already returned a valid action handle.

1.2. Start a task T explicitly using the job handle (mtapi_task_start). Node Y can wait for completion of T to start other tasks (mtapi_task_wait). Task groups can be used to wait for any or all tasks in a group (mtapi_group_wait_all, mtapi_group_wait_any). Multi-instance tasks can be used, equivalently to a parallel region in OpenMP on shared memory systems (e.g., if X has several cores), or to identical MPI processes on a distributed system.

Scenario 2 - Sequential Tasks: The order of task execution is managed by queues provided by the MTAPI runtime system. Tasks are scheduled in the same order as they were enqueued and task execution does not overlap with execution of other tasks enqueued in the same queue. However, this constraint can be weakened by the runtime system implementation, as queue management allows implementation-specific scheduling attributes (see runtime system documentation for precise queue support):

1.3. Same as 1.1.

1.4. Create a queue object Q on a node Z and attach the job handle of A to Q (*mtapi_queue_create*). Obtain a queue handle of Q on node Y, if Y≠Z (*mtapi_queue_get*).

1.5. Enqueue all tasks in Q in the order they should be executed sequentially (*mtapi_task_enqueue*). Node Y can wait for completion of a single task T or for a group of tasks (*mtapi_group_wait_all*, *mtapi_group_wait_any*).

Scenario 3 - Load Balancing (multiple-implementation) Tasks: Synchronization and timing of tasks is done explicitly by the application running on Y. Jobs are executed by one of several nodes providing an action implementing the same job (see runtime system documentation for precise load balancing information):

1.6. Create N action objects for the N nodes capable of computing a job J (*mtapi_action_create*) with the same job identifier. Obtain a job handle on node Y (*mtapi_job_get*).

1.7. Start a task T with the job handle for J (*mtapi_task_start*, *mtapi_task_enqueue*). The MTAPI runtime system selects the action to be executed (i.e., the node that executes the job) depending on the system load.

MTAPI API Excerpts

All API functions are reentrant, i.e., they can be interrupted, reentered, and correctly resumed. For all excerpts, see the complete MTAPI standard for details.

void mtapi_initialize()	Initialize and retrieve system information.
void mtapi_finalize()	Cancel all running tasks, blocks on their completion, finalizes the MTAPI environment.
action_hndl mtapi_action_create()	Create an action from an action function and register it at the runtime with a job ID.
void mtapi_action_delete()	Unregister an action.
mtapi_job_hndl_t mtapi_job_get()	Get a job handle on the same node or on a remote node.
task_hndl mtapi_task_start()	Start a task executing one the action implementing the job passed with as an argument.
mtapi_queue_hndl_t mtapi_queue_create()	Create a queue for a job
task_hndl mtapi_task_enqueue()	Enqueue a task for execution in a queue.