OCR
The Open Community Runtime Interface

Version 1.2.0 (Candidate Release), December, 2016

Editors: Tim Mattson, Romain Cledat

Copyright © 2016 OCR working group.
Permission to copy without fee all or part of this material is granted, provided the OCR working group copyright notice and the title of this document appear.

This page intentionally left blank

Contents

Acknowledgements

The OCR specification is an ongoing and collaborative effort in which multiple people have participated. The following is a partial list of contributors, in alphebetical order as well as the company or institution they represented at the time of their contribution:

Jorge Bellon Castro, Intel Corporation and Barcelona Supercomputing Center
Zoran Budimlic, Rice University
Vincent Cavé, Rice University and Intel Corporation
Sanjay Chatterjee, Intel Corporation
Romain Clédat, Intel Corporation
Jiri Dokulil, University of Vienna
Roger Golliver, Rice University
Min Lee, Intel Corporation
Timothy Mattson, Intel Corporation
Sri Raj Paul, Rice University
Nick Pepperling, Intel Corporation
Vivek Sarkar, Rice University
Bala Seshasayee, Intel Corporation
Rob van der Wijngaart, Intel Corporation
Nick Vrvilo, Rice University

1. Introduction

Extreme scale computers (such as proposed Exascale computers) contain so many components that the aggregate mean-time-between-failure (MTBF) is small compared to the runtime of an application. Programming models (and supporting compilers and runtime systems) must therefore support a variety of features unique to these machines:

- The ability for a programmer to express O(billion) concurrency in an application program.
- The ability of a computation to make progress towards a useful result even as components within the system fail.
- The ability of a computation to dynamically adapt to a high degree of variability in the performance and energy consumption of system components to support efficient execution.
- The ability to either hide overheads behind useful computation or have overheads small enough to allow applications to exhibit strong scaling across the entire exascale system.

There are a number of active research projects to develop runtime systems for extreme scale computers. This specification describes one of these research runtime systems: the *Open Community Runtime* or *OCR*.

The fundamental idea behind OCR is to consider a computation as a dynamically generated directed acyclic graph (DAG) [15, 16, 19] of tasks operating on relocatable blocks of data (which we call data blocks in OCR). Task execution is synchronized through the use of events. When the data blocks and events a task depends upon are satisfied, the preconditions for the execution of the task [13] are met and the task will, at some later point, run on the system. OCR tasks are *non-blocking*. This means that once all preconditions of a task have been met, the task will eventually run to completion regardless of the behavior of any other tasks or events.

Representing a computation in terms of an event-driven DAG of tasks decouples the work of a computation from the "units of execution" that carry out the computation. The work of a computation is virtualized giving OCR the flexibility to relocate tasks to respond to failures in the system [18], achieve a better balance of load among the processing elements of the computer, or to optimize memory and energy consumption [3, 5, 7, 11].

Representing the data in terms of data blocks similarly decouples the data in a computation from the computer's memory subsystem. This supports transparent placement and dynamic migration of data across hardware resources.

1.1. Scope

OCR is a vehicle to support research on programming models and runtime systems for extreme scale computers [9, 10]. This specification defines the state of OCR at a fixed point in its development and will continuously evolve as limitations are identified and addressed.

OCR is both a low level runtime system designed to map onto a wide range of scalable computer systems as well as a collection of low level application programming interfaces (API). It provides the capabilities needed to support a wide range of programming models including data-flow (when events are associated with data blocks), fork-join (when events enable the execution of post-join continuations), bulk-synchronous processing (when event trees can be used to build scalable barriers and collective operations), and combinations thereof. While some programmers will directly work with the APIs defined by OCR, the most common use of OCR will be to support higher level programming models. Therefore, OCR lacks high level constructs familiar to traditional parallel programmers such as **`reductions`** and **`parallel for`**[1]. OCR is therefore not designed to be the primary interface for application level programming.

All parallelism must be specified explicitly in OCR; OCR does not extract the concurrency in a program on behalf of a programmer nor does it make any implicit assumption about the ordering of tasks in contrast to, for example, Legion[1] which executes tasks in parallel as long as the appearance of the sequential order specified implicitly by the ordering of the code is maintained.

OCR is designed to handle dynamic task driven algorithms expressed in terms of a directed acyclic graph (DAG). In an OCR DAG, each node is visited only once. This makes irregular problems based on dynamic graphs easier to express. However, it means that OCR may be less effective for regular problems that benefit from static load balancing or for problems that depend on iteration over regular control structures.

OCR is defined in terms of a C library. Programs written in any language that includes an interface to C should be able to work with OCR.

OCR tasks are expressed as event driven tasks (EDTs). The overheads associated with OCR API calls depend on the underlying system software and hardware. On current systems, the overhead of creating and scheduling an event driven task can be fairly high. On system with hardware support for task queues, the overheads can be significantly lower. An OCR programmer should experiment with their implementation of OCR to understand the overheads associated with managing EDTs and assure that the work per EDT is great enough to offset OCR overheads.

OCR is currently a research runtime system, developed as an open-source community project. It does not as yet have the level of investment needed to develop a production system that can be used for serious application deployment.

[1] Reductions can be supported in OCR using an accumulator/reducer approach [4, 14] and **`parallel for`** can be supported in OCR using a fork-join decomposition similar to the cilk_for construct.

1.2. Version numbers

As OCR is evolving, newer versions of the API may break backward compatibility with older versions of the API. To help the programmer navigate these incompatibilities, the version number for the OCR API will follow the following rules:

- An OCR version number is composed of three integers X, Y and Z and represented as $X.Y.Z$.
- API compatibility is guaranteed between any versions that only vary on the last digit. For example, 1.0.1 is compatible with 1.0.5.
- A change in any of the other digits indicates that an API incompatibility exists. The magnitude of this incompatibility is indicated by whether the middle digit changed (smaller change) or the first digit changed (big change).
- OCR provides macros to check the API version implemented as well as any extensions that are supported.

1.3. Glossary

Acquired The state of a data block when its chunk of data is accessible to an OCR object. For example, an EDT must acquire a data block before it can read-from or write-to that data block.

Data Block (DB) The data, used by an OCR object such as an EDT, that is intended for access by other OCR objects. A data block specifies a chunk of data that is entirely accessible as an offset from a starting address.

Dependence A dependence is a link between the post-slot of a source event, data block or EDT and the pre-slot of a destination EDT or event. The satisfaction of the source OCR object's post-slot will trigger the satisfaction of the destination OCR object's pre-slot.

Event Driven Task (EDT) An OCR object that implements the concept of a task. An EDT with N dependences will have N *pre-slots* numbered from 0 to $N-1$ and one post-slot. Each of the *pre-slots* associated with an EDT connects to a single OCR object, while the EDT's single *post-slot* can connect to multiple OCR objects. An EDT transitions to the *runnable* state when all its pre-slots have been satisfied; the pre-slots determine which data blocks, if any, the EDT may access. In other words, an EDT can only read and write to data blocks that are passed along one of its *pre-slot* or that it creates. Once an EDT is runnable, it is guaranteed to eventually run unless the program terminates early.

EDT function The function that defines the code to be executed by an EDT. An EDT function takes as arguments the number of parameters, the actual array of parameters, the number

of dependences and the actual array of dependences. *Parameters* are static 64-bit values known at EDT creation time. *Dependences* are dynamic control or data dependences and are also refered to as the EDT's pre-slots. The parameter array is copied by value when the EDT is created and enters the *available* state. The dependences (namely the array of dependences) are determined at runtime and are fully resolved when the EDT enters the *resolved* state. An EDT function's return type is a GUID. The GUID returned will be passed along to the EDT's post-slot.

EDT template An OCR object from which an EDT instance is created. The EDT template stores meta-data related to the EDT definition: the EDT function, and the number of parameters and dependences available to EDTs instantiated (created) from this template. Multiple EDTs can be created from the same EDT template.

Event An OCR object used as an indirection mechanism between other OCR objects interested in each other's change of state (unsatisfied to satisfied). Events are the main synchronization mechanism in OCR.

Finish EDT A special class of EDT. As an EDT runs, it may create additional EDTs which may themselves create even more EDTs. In the case of a finish EDT, the EDT's post-slot will only be satisfied after the EDTs created within its scope (i.e. its child EDTs and further descendants) complete and satisfy their post-slots or are destroyed prior to becoming runnable. The result is that any OCR object linked to the post-slot of the finish EDT will, by necessity, not become runnable (i.e. be scheduled for execution) until the finish EDT and all EDTs created during its execution have either completed or been destroyed by the user.

Globally Unique ID (GUID) A value generated by the runtime system that uniquely identifies each OCR object. The GUIDs for the OCR objects reside in a global namespace visible to all EDTs. Creation and management of GUIDs is managed by the OCR runtime by default, though user input can also be used to influence GUID allocation, if so desired.

Latch Event A special type of event that propagates a satisfy signal to its post-slot when it has been satisfied an equal number of times on each of its two pre-slots. In other words, if you imagine a monotonically increasing counter initially set to zero, on each of the two pre-slots, the latch event's post-slot will be satisfied if and only if both monotonically increasing counters are non-zero and equal. Note that once the latch event's post-slot is satisfied, satisfaction on the latch event's pre-slots will result in undefined behavior; the latch event will therefore only satisfy its post-slot at most one time.

Link A dependence between OCR objects typically expressed as a connection between the post-slot of one OCR object and a pre-slot of another. For example if there is a path between the post-slot of a data block and the pre-slot of an EDT, the data block is said to be "linked to the EDT".

OCR object An object managed by OCR. *EDTs*, *events*, *EDT templates*, and *data blocks* are the most frequently encountered examples of OCR objects. Each OCR object has a

unique identifier, or GUID.

OCR program A program that is conformant to the OCR specification. Statements in the OCR specification about the OCR program only refer to behaviors associated with the constructs that make up OCR. For example, if an OCR program were to use a parallel programming model outside of OCR, that program is no longer a purely conformant OCR program and its behavior could no longer be defined by OCR.

Released The state of a linked data block that is no longer accessible by a certain OCR object. For example, after an EDT has finished all of its modification to a data block and it is ready to make those modifications accessible by other EDTs, it must release that data block.

Slot Positional end point for a dependence. An OCR object has one or more slots. Exactly one slot is a *post-slot*. This is used to communicate the state of the OCR object to other OCR objects. The other zero or more slots are *pre-slots*, which are used to manage input dependences of the OCR object. A slot can be:

- Unconnected: there are no links connecting to the slot;
- Connected: a link attaches a source's post-slot to a destination pre-slot.

A slot in the *connected* state can be:

- Satisfied: the source of the link has been triggered;
- Unsatisfied: the source of the link has not been triggered.

Task A non-blocking set of instructions that constitute the fundamental "unit of work" in OCR. By "non-blocking" we mean that once all preconditions on a task are met, the task is runnable and it will execute at some point, regardless of what any other task in the system does. The concept of a task is realized by the OCR object "Event Driven Task" or EDT.

Trigger This term is used to describe the action of either a "satisfied" post-slot or of an event whose trigger rule is satisfied. In the former case, when a post-slot on an OCR object is satisfied, it triggers any connected pre-slots causing them to become "satisfied". In the latter case, when an event's trigger rule is satisfied (due to satisfaction(s) on its pre-slot(s)), it satisfies its post-slot. Therefore, for most events, when the event's pre-slot becomes satisfied, this will **a)** trigger the event causing it to satisfy its post-slot and **b)** trigger the dependence link and satisfy all pre-slots connected to the event's post-slot. The conjugated form *triggered* is used as an attributive past participle; that is: "an EDT that has finished executing the code in its EDT function and released its data blocks will satisfy the event associated with its post-slot and become a triggered EDT".

Unit of Execution A generic term for a process, thread, or any other executable agent that carries out the work associated with a program.

Worker The unit of execution (e.g. a process or a thread) that carries out the sequence of instructions associated with the EDTs in an OCR program. The details of a worker are tied to a particular implementation of an OCR platform and are not defined by OCR.

1.4. OCR objects

An OCR object is an entity managed by OCR. Every OCR object has a globally unique ID (GUID) used to identify the object. An OCR program is defined in terms of three fundamental objects:

- *Event Driven Task (EDT)*: A non-blocking unit of work in an OCR program.
- *Data block (DB)*: A contiguous block of memory managed by the OCR runtime accessible to any OCR object to which it is linked.
- *Event*: An object to manage dependences between OCR objects and to define ordering relationships (synchronization) between them.

In addition to these fundamental objects, other OCR objects play a supporting role; making programming more convenient or providing information the OCR runtime can use to optimize program execution:

- *EDT Template* An OCR object used to manage the resources required to create an EDT.

Objects have two well defined states:

1. *Created*: Resources associated with an object and its GUID have been created.
2. *Destroyed*: An object that is destroyed is marked for destruction when the destruction command executes. Some objects are destroyed automatically, when specific criteria is met. A destroyed object, its GUID, and any resources associated with the destroyed object are no longer defined[2].

More details regarding object creation and destruction are provided in Section 1.4.5. Furthermore, data blocks have two additional states:

1. *Acquired*: the data associated with the data block is accessible to the acquiring OCR object.
2. *Released*: The object is no longer accessible to the OCR object that had earlier acquired it.

1.4.1. Dependences, links and slots

An OCR program is defined as a directed acyclic graph (DAG) with EDTs, data blocks and events as nodes and edges that define *links* between objects. A link is a dependence between OCR objects. The links are defined in terms of *slots* on the OCR object which define an end point for a

[2] As an optimization, the runtime may choose to reuse the same physical object for different logical objects [12, 17]. Similarly GUIDs may be reused for objects that have non-overlaping live ranges.

dependence: the “source” end-point is the source object’s *post-slot* and the “destination” end-point is the destination object’s *pre-slot*.

1.4.1.1. Slots

Post and pre-slots can be in one of three states:

- **Unconnected** when no link is connected to the slot;
- **Connected and unsatisfied** when a link is connected to the slot but the condition that triggers the slot has not been satisfied;
- **Connected and satisfied** when a link is connected to the slot and the condition that triggers the slot has been met. Specifically, a pre-slot is satisfied when the post-slot it is linked to becomes satisfied. A post-slot is satisfied when the OCR object it is attached to is triggered. Data blocks are always considered triggered (conceptually, the data is always ready), EDTs become triggered after they finish executing and release their data blocks and the trigger rules for events are explained in Section 1.4.3.

Initially, all slots are **unconnected**. Note that post-slots can also be **satisfied** but **unconnected**. This happens when, for example, an EDT completes but no other OCR object is waiting for its completion.

1.4.1.2. Dependence

The source of a dependence is always an OCR object’s post-slot. Event, data blocks and EDTs each have a single *post-slot*. A single post slot can be connected to multiple pre-slots thereby allowing the satisfaction of multiple dependences (fan-out).

EDTs and events also have an optional set of *pre-slots*. A pre-slot defines an incoming dependence: for an EDT, this translates to a pre-condition for its execution; for an event, this translates to a pre-condition for its satisfaction.

1.4.1.2.1. Data dependence When a slot is satified, it can optionally be associated with a data block. This mechanism is used to express data dependences in OCR. A data block’s post-slot is associated with the data block itself. An EDT’s post-slot is associated with the data block whose GUID is returned by the EDT. An event’s pre-slot can be satisfied and associated with a data block; that data block will then be passed to the event’s post-slot (except for latch events).

As an example, consider a producer consumer relationships between a pair of EDTs. The post-slot of the producer EDT is linked to the pre-slot of the consumer EDT. When the producer finishes its work and updates the data block it wishes to share, it associates that data block with its post-slot and changes the post-slot’s state to “satisfied”. This triggers the link between the producer and the

consumer making the data block available to the consumer who can now safely use the data block from the producer.

1.4.1.2.2. Control dependence

In OCR, control dependences are equivalent to data dependences where the data block associated with the dependence is an empty one.

1.4.2. Event Driven Task (EDT)

The fundamental unit of work in OCR is the *Event Driven Task* or *EDT*. When all pre-conditions on an EDT have been met it becomes a runnable EDT. Later when its input data blocks are acquired, the EDT is ready to execute. The OCR runtime guarantees that a runnable EDT will execute at some point[3] and once running, the EDT will progress to its terminal state and cannot be halted by the action of other OCR objects; hence the execution of an EDT is said to be *non-blocking*.

The work carried out by an EDT is defined by the *EDT function*. The EDT function prototype and return values are defined in more detail in the OCR API (see Section 2.8). Briefly, an EDT function takes as input:

- An array of 64-bit values called parameters. These parameters must be known at the time the EDT is created and are copied by value at that time.
- A number of dependences on other OCR objects (typically events and data blocks).

An EDT function also returns a GUID which, if non-NULL, can refer to another OCR object which will be used to satisfy the EDT's post-slot. The GUID returned by an EDT must be either a data block GUID or the NULL GUID.

When the OCR API is used to create an EDT (using the **`ocrEdtCreate()`** function) one or two GUIDs are returned. The first (always returned) is the GUID for the EDT itself. The second (returned only on programmer request) is the GUID of the event implied by the post-slot of the EDT[4] When the OCR function returns a data block, the GUID of that data block is used to satisfy this implied event.

Using a post-slot in a link to another object is just one method to trigger other OCR objects. OCR includes the **`ocrEventSatisfy()`** function to trigger other OCR objects through explicitly created dependence links. The OCR runtime, however, is allowed to defer all event satisfactions to the end of the EDT. This is an important performance optimization designed into OCR. This is also consistent with the intent of OCR to define the state of an evolving computation by the versions of data blocks and a log of the EDTs that have completed. This implies that an OCR programmer should ideally treat EDTs as small units of work that execute with transactional semantics.

[3] Unless the entire program is terminated while there are still runnable EDTs.

[4] It is important to note that although, semantically, an EDT can be the source of a dependence, when adding a dependence, the programmer must use the GUID of the associated event as the source.

OCR also defines a special type of EDT; the *finish EDT*. An EDT will always execute asynchronously and without blocking once all of its pre-conditions have been met. A finish EDT, however, will not trigger its post-slot until all EDTs launched within its scope (i.e. its child EDTs and all its transivitely created grand-child EDTs) have completed. The finish EDT still executes asynchronously and without blocking. The implied event associated with the post-slot of a finish EDT is a *latch event*, i.e. it is connected to the post-slots of all EDTs created within its scope and does not trigger until they have all finished. Currently, the returned value of finish EDTs is ignored and no data block can be passed through the post-slot of a finish EDT.

For both normal and finish EDTs, the EDT is created as an instance of an *EDT template*. This template stores metadata about EDTs created from the template, optionally defines the number of dependences and parameters used when creating an instance of an EDT, and is a container for the function that will be executed by an EDT. This function is called the *EDT function*.

1.4.3. Events

An event is an OCR object used to coordinate the activity of other OCR objects. As with any OCR object, events have a single post-slot. Events may also have one or more pre-slots; the actual number of which is determined by the type of event.

The post-slot of an event can be connected to multiple OCR objects by connecting the single post-slot to the pre-slots of other OCR objects. When the conditions are met indicating that the event should trigger (according to the *trigger rule*), the event sets its post-slot to *satisfied* therefore establishing an ordering relationship between the event and the OCR objects linked to the event. Events therefore play a key role in establishing the patterns of synchronization required by a parallel algorithm [8].

When an event is satisfied, it can optionally include an attached data block on its post-slot. Hence, events not only provide synchronization (control dependences) but they are also the mechanism OCR uses to establish data flow dependences. In other words, a classic data flow algorithm defines tasks as waiting until data is “ready”. In OCR this concept is implemented through events with attached data blocks.

Given the diversity of parallel algorithms, OCR has defined several types of events:

1. *Once event*: The event is automatically destroyed on satisfaction. Any object that has the Once event as a pre-condition must already have been created and linked by the time the Once event is satisfied.
2. *Idempotent event*: The event exists until explicitly destroyed by a call to **`ocrEventDestroy()`**. It is satisfied once and subsequent attempts to satisfy (i.e. trigger) the event are ignored.
3. *Sticky event*: The event exists until explicitly destroyed with a call to **`ocrEventDestroy()`**. It is satisfied once and subsequent attempts to satisfy (i.e. trigger) the event result in an error code being returned when trying to satisfy the event.

4. *Latch event*: The latch event has two pre-slots and triggers when the conditions defined by the latch trigger rule are met. The event is automatically destroyed once it triggers; in this regard, it is similar to a *once event*.

1.4.3.1. Event trigger rules

Events "trigger" when the appropriate *trigger rule* is met. The default trigger rule for events is to trigger their post-slot when any of their pre-slot becomes satisfied. Any data block associated with the pre-slot is also passed to the post-slot.

The trigger rule for a latch event is somewhat more complicated. The latch event has two pre-slots; an increment slot and a decrement slot. The latch event will trigger its post-slot when the event receives an equal but non-zero number of satisfy notifications on each of the pre-slots. Once a latch event triggers, any subsequent triggers on the pre-slots of the latch event are undefined. For regular events, when it is triggered with a data block, the GUID of that data block is passed along through the post-slot of the event. For a latch event, however, the GUID of a data block that triggers a pre-slot, if any, is ignored.

1.4.4. Data blocks

Data blocks are OCR objects used to hold data in an OCR program. A data block is the only way to store data that persists outside of the scope of a collection of EDTs. Hence, data blocks are the only way to share data between EDTs. The data blocks are identified by their GUIDs and occupy a shared name space of GUIDs. While the name space is shared and globally visible, however, an EDT can only access **a)** data blocks passed into the EDT through a pre-slot or **b)** a data block that is created inside the body of the EDT.

When a data block is created, the default behavior is that the EDT that created the data block will also acquire the data block. Optionally, an EDT can create a data block on behalf of another EDT. In this case, a programmer can request that the data block is created, but not acquired by the EDT.

Conceptually, data blocks are contiguous chunks of memory that have a start address and a size. They have the following characteristics:

- all memory within the data block is accessible from the start address using an offset, meaning an EDT can manipulate the contents of a data block through pointers.
- The contents of distinct data blocks are guaranteed to not overlap.
- The pointer to the start of a data block is only valid between the acquire of the data block (implicit when the EDT starts) and the corresponding **`ocrDbRelease()`** call (or the end of the acquiring EDT, whichever comes first)

Data blocks can be explicitly connected to other OCR objects through the OCR dependence API (see Chapter 2.9). The more common usage pattern, however, is to attach data blocks to events and

pass them through the directed acyclic graph associated with an OCR program to support a data-flow pattern of execution.

Regardless of how the data blocks are exposed among a collection of EDTs, a program may define constraints over how data blocks can be used. This leads to several different modes for how an EDT may access a data block. The mode is set when the OCR dependences API is used to dynamically set dependences between a data block and an EDT. Currently, OCR supports five modes:

1. *Read-Write* (default mode): The EDT may read and write to the data block. Note that no exclusive access is enforced; multiple EDTs may write to the same data bock at the same time. It is the responsibility of the programmer to ensure that when multiple writers are writing to a *read-write* data block, appropriate synchronization (in the form of dependences) is included to assure that the writes do not conflict. It is legal for an OCR program to contain data races. Section 1.6 contains more information.

2. *Exclusive write*: The EDT may read and write to the data block and OCR will enforce that, at a given time, it is the only EDT writing to the data block. When the writing EDT releases the data block, another EDT may acquire it and write to it. Given the memory model described in Section 1.6, the writes of multiple EDTs all acquiring a data block in exclusive write will be sequential but the only ordering between said EDTs is determined by the explicit dependences. In other words, if both EDTs A and B are runnable and access a data block D in exclusive write mode, A and B can execute in any order and the only guarantee is that there will be no overlap between the two following intervals: start of A to the release of D by A (or the end of A, whichever comes first) and the start of B to the release of D by B (or the end of B, whichever comes first).

3. *Read only*: The EDT will only read from the data block. OCR does not restrict the ability of other EDTs to write to the data block, even if the writes from one EDT might overlap with reads by the EDT with *read only* access. If an EDT writes to a data block it has acquired in *read only* mode, the results of those writes are undefined should other EDTs later acquire the same data block.

4. *Constant*: The EDT will only read from the data block and OCR will ensure that once the data block is acquired, writes from other EDTs will not be visible to the EDT acquiring the data block in constant mode. This is the distinction between this mode and the read only mode. If an EDT writes to a data block it has acquired in *constant* mode, the results of those writes are undefined should other EDTs later acquire the same data block.

5. *NULL*: The EDT will not access the data block either to read or write. This mode allows the user to convert a data dependence into a pure control dependence. This is useful, if, for example, two EDTs A and B depend on the completion of an event E but only one, A for example, caresis to let the runtime know that the EDT does not care about the data passed to the EDT (through an event for example) and has the effect of converting a potential data dependence into a pure control dependence. Note that the GUID of the data block, if any, is still passed to the EDT.

1.4.5. Object lifetime

OCR objects are created by the appropriate OCR API call, for example **`ocrEventCreate`**. Even though the runtime may defer the actual creation of the object, it ensures that it appears to be valid to all subsequent OCR API calls. Objects are destroyed either automatically or using an OCR API call, such as **`ocrEventDestroy`**. Again, the runtime may actually destroy the object at a later point in time, but the object cannot be used after it has been formally destroyed.

In this Section, we detail when a user can assume an object is destroyed.

1.4.5.1. Data blocks

Data blocks are explicitly destroyed using the **`ocrDbDestroy`** call. The runtime will delay the actual destruction of the data block and associated resources until all EDTs that have acquired the data block prior to the call to **`ocrDbDestroy`** have released it. In practice, all EDTs whose start happens before the call to **`ocrDbDestroy`** are guaranteed to be able to use the data block. For example, if EDT *A* and EDT *B* both satisfy, at their beginning, two dependences that allow EDT *C* to run, EDT *C* can safely destroy any data block that *A* and *B* are using.

1.4.5.2. Events

Once and latch events are destroyed automatically after they are triggered and the destruction occurs before any pre-slots connected to the event's post-slot are satisfied. Sticky and idempotent events are destroyed explicitly and the runtime guarantees that any **`ocrAddDependence`** call adding a dependence from the event to another object that happened before the destruction call will be satisfied if the event destroyed is triggered. In particular, an event can be safely destroyed by any EDT that depends on its satisfaction provided all dependences were added prior to the triggering of the event.

1.5. Execution Model

OCR is based on an asynchronous task model. The work of an OCR program is defined in terms of a collection of tasks organized into a directed acyclic graph (DAG) [15, 16, 19]. Task execution is managed by the availability of data (the "data blocks") and events; hence the reason the tasks are called "Event Driven Tasks" or EDTs.

An OCR program executes on an abstract machine called the *OCR Platform*. The OCR platform is a resource that can carry out computations. It consists of:

- A collection of network connected nodes where any two nodes can communicate with each other.

- Each node consists of one or more processing elements each of which may have its own private memory[5].
- Workers that run on the processing elements to execute enqueued EDTs.
- A globally accessible shared name space of OCR objects each denoted by a globally unique ID (GUID).

OCR is designed to be portable and scalable, hence, the OCR Platform places minimal constraints on the physical hardware.

The OCR program logically starts as a single EDT called **`mainEDT()`**. In other words, the programmer does not provide a **`main()`** function. The OCR runtime system creates the **`main()`** function on the programmer's behalf to set up the OCR environment and then calls the user provided **`mainEDT()`**. The expected function prototype for the **`mainEDT()`** is described in section 2.3.

The DAG corresponding to the executing program is constructed dynamically and completes when the **`ocrShutdown()`** or **`ocrAbort()`** function is called. This rather simple model can handle a wide range of design patterns including branch and bound, data flow, and divide and conquer.

To understand the execution model of OCR, consider the discrete states and transitions of an executing EDT as defined in figure 1.1. An EDT is created and once its GUID is available for use in API functions, the EDT is said to be *Available*. At some point, the dependences are fully defined (all pre-slots of the EDT are "connected") for the EDT and it becomes *Resolved*. Note that the transition from *Available* to *Resolved* is not called out as a named transition. This implies that it is not generally possible for the system to set a distinct time-stamp corresponding to when the transition occurred. In this case, the transition is un-named because dependences may be added dynamically up until the EDT *Launch* transition. At this point the EDT is *Runnable*.

Once an EDT is runnable, it will execute at some point during the normal execution of the OCR program. At some point all data blocks linked to an EDT will be acquired and the EDT becomes *Ready*. The EDT and any resources required to support its execution are then submitted to *workers* [6] which execute the tasks on the processing elements within the OCR platform. The workers and the data-structures used to store tasks waiting to execute (i.e. work-pools) are a low level implementation detail not defined by the OCR specification. When reasoning about locality and load balancing, programmers may need to explicitly reason about the behavior of the workers [2], but they do not hold persistent state visible to an OCR program and are logically opaque to OCR constructs. The scheduler inside the implementation of OCR will then schedule the EDT for execution and the EDT *Starts* to execute and becomes a *Running* EDT.

Normal EDT execution continues until the EDT function returns. The EDT undergoes a *Finish* transition and the EDT is in the *End* state. At some point the EDT will release the data blocks associated with the EDTs execution and the EDT enters the *Released* state. At this point, any changes made to data blocks will be available for use by other OCR objects. Later the EDT will mark its post-slot as satisfied to *Trigger* the event associated with the EDT; thereby becoming a

[5] By "private" we mean a memory region that is not accessible to other processing elements.

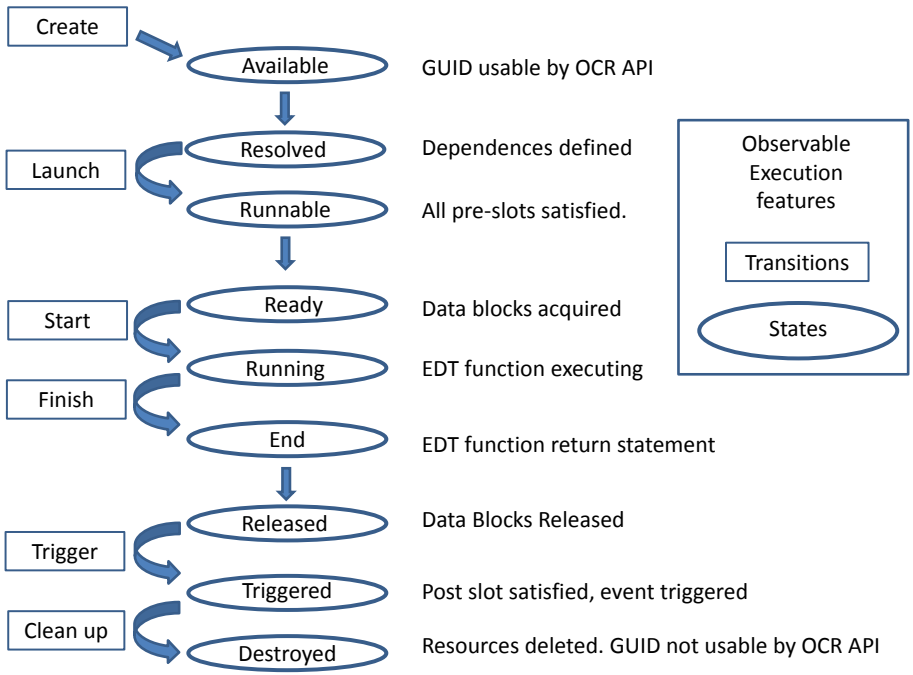

Figure 1.1.: Observable execution features.

Triggered EDT. At some later point the system will *Clean-up* resources used by the EDT (including its GUID) and the EDT is destroyed.

Since an EDT is non-blocking, once it becomes *Runnable* it will run on the OCR platform at some point in the future. During its run:

- The EDT can only access data blocks that have been passed in through its pre-slots as well as any data blocks that the EDT creates internally. This means that before an EDT starts, the OCR runtime knows all the data blocks that will be accessed (minus the ones created within the EDT).
- The EDT can call into the runtime to create and destroy data blocks, EDTs and events.
- The EDT can create *links* between the various OCR software constructs, termed *dependences*. This is accomplished through the **`ocrAddDependence()`** function of the OCR API. The following types of dependences can be created:
  - *Event to Event*: The destination event's pre-slot is chained directly to the source event's post-slot. For all events but the latch event, this means that the triggering of the source event will trigger the destination event.
  - *Event to EDT*: One of the destination EDT's pre-slots is chained directly to the source event's post-slot. When the source event is triggered, this will satisfy the EDT's pre-slot. If a data block was associated with the triggering of the source event, that data block will be made available to the EDT in the dependence array in the position of the pre-slot. This is a "control and data" dependence. In the other case, no data block will be made available and the dependence is akin to a pure control dependence.
  - *Data block to Event*: Adding a dependence between a data block and an event will result in the satisfaction of the event with the data block; however, the satisfaction of the event may be deferred and may happen asynchronously.
  - *Data block to EDT*: Directly adding a dependence between a data block and an EDT (a pure data-dependence) immediately satisfies the EDT's pre-slot and makes the data block available to the EDT in the dependence array in the position of the pre-slot.
- The EDT cannot perform any synchronization operations that would cause it to block inside the body of the task (i.e. the EDT must be non-blocking). The only mechanism for synchronization within OCR is through the events that link OCR objects, which are explicit to the runtime.

A computation is complete when an EDT terminates the program (e.g. with a call to **`ocrShutdown()`**). Typically, the EDT that terminates the program is the last EDT in the program DAG, and the programmer has assured that all other EDTs in the DAG have completed execution before the function to terminate the program is called.

Since the OCR runtime creates the **`main()`** function, the programmer does not need to manage the low level details of initializing and cleanly shutting down OCR.

With both data and tasks conceptually decoupled from their realization on a computer system, OCR has the flexibility to relocate tasks and data to respond to failures in the system, achieve a better

balance of load among the processing elements of the computer, or to optimize memory and energy consumption [3, 5, 7, 11]. This requires that the state of an OCR program can be defined strictly in terms of which tasks have completed their execution and the history of updates to data blocks. By saving a log of updates to data blocks relative to the tasks that have completed execution, the system can recover the state of a computation should components of the system fail. This requires, however, that EDTs execute with transactional semantics.

1.6. Memory Model

A memory model defines the values that can be legally observed in memory when multiple units of execution (e.g. processes or threads) access a shared memory system. The memory model provides programmers with the tools they need to understand the state of memory, but it also places restrictions on what a compiler writer can do (e.g. which aggressive optimizations are allowed) and restrictions on what a hardware designer is allowed to do (e.g. the behavior of write buffers).

1.6.1. Definitions

To construct a memory model for OCR, we need to present a few definitions. The operations inside an EDT execute in a non-blocking manner. The order of such operations are defined by the *sequenced-before* relation defined by the host C programming language.

When multiple EDTs are running, they execute asynchronously. Usually, a programmer can make few assumptions about the relative orders of operations in two different EDTs. At certain points in the execution of EDTs, however, the OCR program may need to define ordering constraints. These constraints define *synchronized-with* relations.

The "transitive closure" of sequenced-before operations inside each of two EDTs combined with the synchronized-with relations between two EDTs defines a *happens-before* relationship. For example:

- if **`A`** is sequenced-before **`B`** in EDT1
- if **`C`** is sequenced-before **`D`** in EDT2
- and **`B`** is synchronized-with **`C`** in EDT2
- then **`A`** happens-before **`D`**.

These basic concepts are enough to define the memory model for OCR.

1.6.2. OCR memory model

OCR provides a relatively simple memory model. Before an EDT can read or write a data block, it must first *acquire* the data block. This is not an exclusive relationship by which we mean it is possible (depending on the mode of the data block in question) for multiple EDTs to acquire the same data block at the same time. When an EDT has finished with a data block and it is ready to expose any modifications to the data block to other EDTs, it must *release* that data block. Acquiring data blocks happens implicitly only at the start of an EDT while releasing data blocks can happen either implicitly at the end of an EDT or explicitly through an API call.

> When an EDT calls an OCR function that releases a data block, the OCR runtime must ensure that all loads and stores to the data block by that EDT complete before the data block is released. Furthermore, the OCR runtime must ensure that the release completes before returning from the release function call.

The only way to establish a synchronized-with relation is through the behavior of events. If the pre-slot of EDT2 is connected to the post-slot of EDT1, then EDT2 waits for event associated with the post-slot of EDT1 to trigger. Therefore, the satisfy event from EDT1 synchronizes-with the triggering of the pre-slot of EDT2. We can establish a happens-before relationship between all operations in EDT1 and any operation in EDT2 if we define the following rule for OCR.

> The OCR runtime must ensure that an EDT completes the release of all of its data blocks before it marks its post-event as satisfied.

An EDT can use data blocks to satisfy events in the body of the task in addition to the event associated with its post-slot. We can reason about the behavior of the memory model and establish happens-before relationships if we define the following rule.

> Before an EDT calls a function to satisfy an event, for any data block potentially exposed to other EDTs by that event satisfaction, all writes to that data block must complete and the data block must be released before the event is satisfied.

Without this rule we cannot assume a release operation followed by satisfying an event defines a sequenced-before relationship that can be used to establish a happens-before relation.

The core idea in the OCR memory model is that happens-before relationships are defined in terms of events (the only synchronization operation in OCR) and the release of OCR objects (such as data blocks). This is an instance of a *Release Consistency* memory model which has the advantage of being relatively straightforward to apply to OCR programs.

The safest course for a programmer is to write programs that can be defined strictly in terms of the release consistency rules. OCR, however, lets a programmer write programs in which two or more EDTs can write to a single data block at the same time (or more precisely, the two EDTs can issue writes in an unordered manner). This may result in a data race in that the value that is ultimately stored in memory depends on how a system chooses to schedule operations from the two EDTs.

Most modern parallel programming languages state that a program that has a data race[6] is an illegal program and the results produced by such a program are undefined. These programming models then define a complex set of synchronization constructs and atomic variables so a programmer has the tools needed to write race-free programs. OCR, however, does not provide any synchronization constructs beyond the behavior of events. This is not an oversight. Rather, this restricted synchronization model helps OCR better scale on a wider range of parallel computers.

OCR, therefore, allows a programmer to write legal programs that may potentially contain data races. OCR deals with this situation by adding two more rules. In both of these rules, we say that address range A and B are non-overlapping if and only if the set A_1 of 8-byte[7] aligned 8-byte words fully covering A and the set B_1 of 8-byte aligned 8-byte words fully covering B do not overlap. For example, addresses $0x0$ and $0x7$ overlap (assuming byte level addressing) whereas $0x0$ and $0x8$ do not. The first rule deals with the situation of multiple EDTs writing to a data block with non-overlapping address ranges.

> If two EDTs write to a single data block without a well defined order, if the address ranges of the writes do not overlap, the correct result of each write operation must appear in memory.

This behavior may seem obvious making it trivial for a system to support. However, when addresses are distinct but happen to share the same cache lines or when aggressive optimization of writes occur through write buffers, an implementation could mask the updates from one of the EDTs if this rule were not defined in the OCR specification.

The last rule addresses the case of overlapping address ranges. Assume that a system writes values to memory at an atomicity of N-bytes. This defines the fundamental store-atomicity for the system.

> If two EDTs write to a single data block without a well defined order, if the address ranges of the writes overlap, the results written to memory must correspond to one of the legal interleavings of statements from the two EDTs at an N-byte aligned granularity. Overlapping writes to non-aligned or smaller than N-byte granularity are not defined.

For systems that do not provide store-atomicity at any level, N would be 0 and the above rule states that unordered writes to overlapping address ranges are undefined. This rule is the well known *sequential consistency* rule. It states that the actual result appearing in memory may be nondeterministic, but it will be well defined and it will correspond to values from one EDT or the other.

Release consistency remains the safest and best approach to use in writing OCR programs. It is conceivable that some of the more difficult rules may be relaxed in future versions of OCR (especially the sequential consistency rule), but the relaxed consistency model will almost assuredly always be supported by OCR.

---

[6] A *data race* occurs when loads and stores by two units of execution operate on overlapping memory regions without a synchronized-with relation to order them

[7] The reference to "8-byte" words assumes the processing elements utilize a 64-bit architecture. For other cases, all references to an 8-byte word in this specification must be adjusted to match the architecture of the processing elements.

Any memory in OCR that can be accessed by multiple EDTs resides in data blocks. As discussed in Section 1.4.4 there are four access modes for the data blocks in OCR, namely *Read-Write*, *Exclusive write*, *Read only*, and *Constant*.

1.7. Organization of this document

The remainder of this document is structured as follows:

- Chapter 2 defines the OCR Application Programming Interface.
- Appendix A contains a set of pedagogical examples.
- Appendix B contains a set of proposed OCR Extensions.
- Appendix C contains notes specific to current implementations of OCR.
- Appendix D documents the "change history" for OCR and this specification.

2. The OCR API

This chapter describes the syntax and behavior of the OCR API functions, and is divided into the following sections:

- The core types, macros and error codes used in OCR. (Section 2.1 on page 20)
- A description of OCR main entry point: **`mainEdt`**. (Section 2.3 on page 25)
- Supporting functions. (Section 2.4 on page 25)
- Functions to create, destroy, and otherwise manage the contents of OCR data blocks. (Section 2.6 on page 31)
- Functions to manage events in OCR. (Section 2.7 on page 36)
- Functions to create and destroy tasks in OCR. (Section 2.8 on page 40)
- Functions to manage dependences between OCR objects. (Section 2.9 on page 45)

Types, constants, function prototypes and anything else required to use the OCR API are made available through the **`ocr.h`** include file. You do not need to include any other files unless using extended or experimental features (described in the appendices).

2.1. OCR core types and macros

The OCR Application Programming Interface (API) is defined in terms of a C language binding. The functions comprising the OCR API make use of a number of basic data types defined in the include file **`ocr.h`**.

Base low-level types The lowest level data types are defined in terms of the following C typedef statements:

- **`typedef uint64_t u64`** 64-bit unsigned integer;
- **`typedef uint32_t u32`** 32-bit unsigned integer;
- **`typedef uint16_t u16`** 16-bit unsigned integer;

- **`typedef uint8_t u8`** 8-bit unsigned integer;
- **`typedef int64_t s64`** 64-bit signed integer;
- **`typedef int32_t s32`** 32-bit signed integer;
- **`typedef int8_t s8`** 8-bit signed integer;
- **`typedef u8 bool`** 8-bit boolean.
- **`ocrEdtDep_t`**: Type of dependences as passed into the EDTs. It contains at least two fields that can be used by the programmer: **`guid`** which contains the GUID of the data block passed in and **`ptr`** which contains a pointer to the aforementioned data block.

OCR opaque types In addition to these low level types, OCR defines a number of opaque data types to manage OCR objects and to interact with the OCR environment:

- **`ocrGuid_t`**: Opaque handle used to reference all OCR objects. An **`ocrGuid_t`** is truly opaque and the user cannot assume that it is of a given size;

OCR version utilities OCR provides macros to determine the API implemented and the extensions that are enabled:

- **`OCR_VERSION`**: A string representing the API version implemented.
- **`OCR_VERSION_GET_MAJOR(vers)`**: Extract the unsigned integer representing the major revision of the OCR API. This is the first digit of the version string.
- **`OCR_VERSION_GET_MINOR(vers)`**: Extract the unsigned integer representing the minor revision of the OCR API. This is the middle digit of the version string.
- **`OCR_VERSION_GET_PATCH(vers)`**: Extract the unsigned integer representing the patch number of the OCR API. This is the last digit of the version string.
- **`OCR_VERSION_EXTENSION_BITMAP`**: A bitmap representing the extensions enabled. The file "ocr-version.h" defines the possible extensions that are encoded in this bitmap.

Other constants The C API for OCR makes use of a set of basic macros defined inside **`ocr.h`**:

- **`#define true 1`**;
- **`#define TRUE 1`**;
- **`#define false 0`**;
- **`#define FALSE 0`**;
- **`#define NULL_HINT`**: A NULL pointer to an ocrHint_t.

- **`#define NULL_GUID`**: A reserved GUID value, used to indicate the absence of an OCR object. This value is roughly analogous to a NULL pointer; however, it cannot be assumed to be equal to zero.
- **`#define UNINITIALIZED_GUID`**: A reserved GUID value, used as a placeholder to indicate that a value is uninitialized.
- **`#define ERROR_GUID`**: A reserved GUID value, used to indicate that an error has occurred.

2.2. OCR API conventions

In OCR, most APIs are allowed to be *deferred* or *executed asynchronously* by the runtime implementation but all implementations guarantee that the effects of the execution of the API calls by the runtime will be identical to the non-deferred and sequential execution of the calls as specified by the user. Note that for APIs that return a value to the user (for example a GUID, a pointer, etc.), all runtime implementations guarantee that the values returned are valid; in other words, a portion of the call may be deferred but the user can rely on the returned values.

2.2.1. Conventions

In this chapter, the OCR APIs are discussed in detail. API calls not related to support functions (Section 2.4) or GUID management (Section 2.5) all follow certain conventions described below

API names All API names start with "ocr" followed by the type of object the call refers to [1] such as "Edt", "Db" or "Event" followed by the action on that event.

Arguments Arguments that are used as "out" or "in/out" arguments, if any, are listed first in the calls. "Out" or "in/out" arguments are always pointers. "In" arguments are either values or pointers to constants.

Error codes All API calls return an error code with the convention that 0 means the call was successful. Error codes are discussed in more detail in Section 2.2.2.

2.2.2. OCR error codes

The OCR error codes are derived from standard error codes. They are defined internally to limit OCR's dependence on standard libraries. OCR uses the convention that a function's return value is

[1] **`ocrAddDependence()`** being the lone exception to this naming convention.

its error code where a code of zero signifies successful completion. It is, however, important to note that given the potential deferred and asynchronous nature of OCR API calls, not all errors will be reported at the site of the call. This distinction is discussed later in this section. The error codes returned are found in the **`ocr-errors.h`** include file and described in Table 2.2.2.

Error code	Generic Interpretation
OCR_EPERM	Operation not permitted
OCR_ENOENT	No such file or directory
OCR_EINTR	Interrupted OCR runtime call
OCR_EIO	I/O error
OCR_ENXIO	No such device or address
OCR_E2BIG	Argument list too long
OCR_ENOEXEC	Exec format error
OCR_EAGAIN	Try again
OCR_ENOMEM	Out of memory
OCR_EACCES	Permission denied
OCR_EFAULT	Bad address
OCR_EBUSY	Device or resource busy
OCR_ENODEV	No such device
OCR_EINVAL	Invalid argument
OCR_ENOSPC	No space left on device
OCR_ESPIPE	Illegal seek
OCR_EROFS	Read-only file system
OCR_EDOM	Math argument out of domain of func
OCR_ERANGE	Math result not representable
OCR_ENOSYS	Function not implemented
OCR_ENOTSUP	Function is not supported
OCR_EGUIDEXISTS	The objected referred to by GUID already exists
OCR_EACQ	Data block is already acquired
OCR_EPEND	Operation is pending
OCR_ECANCELED	Operation canceled

Immediate versus deferred error Since the OCR APIs can be deferred or executed asynchronously by the underlying implementation, not all error codes can be reported immediately at the API call-site. In the API description in the following sections, the error codes returned are separated into two categories: **a)** the *immediate* error codes that are guaranteed to be returned at the call-site and **b)** the *deferred* error codes that *may* be returned either at the call-site but also later (see the following paragraph for more detail).

Immediate errors are typically errors that result from incorrect arguments. Deferred errors are either returned at the call-site or reported later, depending on the implementation. In other words, if the return code of an API call is zero, this means that either the function completed without any errors or that one of the deferred errors may occur later.

Reporting of deferred errors Currently, if an error occurs in a deferred manner, implementations are allowed to crash but will report the API call that caused the crash (deferred error) by reporting to the user at least the following information:

- File and line-number of the API call that caused the error;
- Error code;
- GUIDs of both the EDT in which the error occurred and the target object for the error (typically the first argument of the API call).

Future revisions of this specification will provide a handler mechanism which would allow a user to determine if an error is fatal or if it can be recovered from.

2.3. OCR entry point: `mainEdt`

An OCR program's single point of entry is the user-defined main EDT (**`mainEdt()`**). This EDT has a function prototype that is identical to any other EDT; its parameters and dependences are however special: the main EDT has no parameters and has a single input data block that encodes the arguments passed to the program from the command line. The arguments can be accessed using **`ocrGetArgc()`** and **`ocrGetArgv()`**.

ocrGuid_t mainEdt(u32 paramc, u64 paramv, u32 depc, ocrEdtDep_t depv[])*

Parameters

in	**paramc**	Parameters count will always be 0.
in	**paramv**	Parameter vector will always be NULL.
in	**depc**	Dependence count will always be 1.
in	**depv**	Dependence vector will have exactly 1 entry containing a data block that encodes the command line arguments.

Returns **`mainEdt`** returns a **`ocrGuid_t`** which is ignored by the runtime. The returned value of an OCR program is set using **`ocrShutdown()`** or **`ocrAbort()`**.

2.4. Supporting functions

OCR provides a set of basic capabilities to support a program execution environment, handled through the following functions.

Functions

- void **`ocrShutdown`**(void)

  Called by an EDT to indicate the normal end of an OCR program.

- void **ocrAbort**(u8 errorCode)

  Called by an EDT to indicate an abnormal end of an OCR program.

- u64 **ocrGetArgc**(void *dbPtr)

  *Retrieves the traditional 'argc' value from **mainEdt()**'s input data block.*

- char * **ocrGetArgv**(void *dbPtr, u64 index)

  *Retrieves the indexth argument from **mainEdt()**'s input data block.*

- u32 **ocrPrintf**(const char *format, ...)

  ***printf()** equivalent for OCR.*

2.4.1. void ocrShutdown( )

The user is responsible for indicating the end of an OCR program using an explicit shutdown call (either this function or **ocrAbort()**). Any EDTs which have not reached the *runnable* state will never be executed. EDTs in the *runnable* or *ready* state may or may not execute. Calling **ocrShutdown()** will properly terminate the runtime; all memory and computing resources will be released. Furthermore, a zero exit code will be returned to the caller of the OCR program indicating a normal termination.

Note

Program behavior after this call is undefined. Specifically:
- The statements in the calling EDT *after* this function call may or may not be executed;
- EDTs in the *runnable* or *ready* state may or may not execute.

Although most programs will choose to call **ocrShutdown()** in the "last" EDT, OCR specifically allows another EDT to call **ocrShutdown()** to support, for example, a computation of the type "find-first-of"; in such a computation, the program can successfully complete even if not all EDTs have executed.

2.4.2. void ocrAbort(u8 *errorCode*)

This function is very similar to **ocrShutdown()** except that it allows for the return of a non-zero value. This function also does not guarantee that the runtime will properly shut itself down and some implementations may choose to crash out to provide, for example, a core dump for debugging. This call should only be used to indicate an abnormal termination.

The abort error code is passed back to the runtime and its handling is implementation dependent

Parameters

in	**errorCode**	User defined error code returned to the runtime.

Description See notes in Section 2.4.1.

2.4.3. u64 getArgc(void * *dbPtr*)

Returns the number of arguments (traditionally called 'argc') passed to the OCR program. The value is extracted from the unique data block passed to **mainEdt**.

Parameters

in	**dbPtr**	Pointer to the start of the argument data block

Returns The number of arguments passed to the OCR program on the command line

Description When starting, the first EDT (called **mainEdt**) is passed a single input data block which encodes the arguments passed to the main program:

- The first 8 bytes encode 'argc';
- The following 'argc' 8-byte values encode the offset from the start of the data block to the arguments;
- The arguments are then appended as NULL terminated character arrays.

2.4.4. char * getArgv(void * *dbPtr* u64 *index*)

Returns the 0-based $index^{th}$ argument passed to the OCR program. The value is extracted from the unique data block passed to **mainEdt**.

Parameters

in	**dbPtr**	Pointer to the start of the argument data block
in	**index**	Index of the argument to extract

Returns A NULL terminated string corresponding to **argv[index]**.

Description See Section 2.4.3 for details.

Note

Attempting to extract an argument with **index** greater or equal to the value returned by **ocrGetArgc** will result in undefined behavior.

2.4.5. u32 ocrPrintf(const char * *fmt* ...)

A platform independent limited **printf** functionality.

Parameters

in	**fmt**	NULL terminated C format string containing the format of the output string
in	**...**	A variable length list of arguments in agreement with **fmt**

Returns This function returns the number of bytes written out as a **u32**.

Description OCR must support a wide variety of platforms including simulators that emulate real systems. Often, core functionality provided by standard C libraries are not available on all platforms, and, as a result, an OCR program cannot depend on these functions. There was one case, however, where it was felt support was critical: basic printf. This function supports basic printing functionality and supports the following format specifiers:

- Strings using **%s**;
- 32-bit integers using **%d**, **%u**, **%x** and **%X**;
- 64-bit integers using **%ld**, **%lu**, **%lx**, **%lX**, and versions with two 'l';
- 64-bit pointers using **%p**;
- Floating point numbers using **%f**, **%e** and **%E**;
- The '#' flag is supported for **%x**, **%lx** and **%llx**;
- Precision modifiers are also supported for **%f**, **%e** and **%E**.

Note

A conforming implementation may limit the number of characters in the output string.

2.5. GUID management

GUIDs are opaque values used to uniquely identify OCR objects. In previous versions of OCR, GUID values could be safely cast to 64-bit values. To preserve the runtime's freedom in encoding information into GUIDs, users can no longer rely on the ability to convert GUIDs to 64-bit values and must instead use the provided functions to manipulate GUIDs.

Functions

- bool **ocrGuidIsNull**(ocrGuid_t guid)

  Checks if the GUID provided is equivalent to NULL_GUID.
- bool **ocrGuidIsUninitialized**(ocrGuid_t guid)

  Checks if the GUID provided is equivalent to UNINITIALIZED_GUID.
- bool **ocrGuidIsError**(ocrGuid_t guid)

  Checks if the GUID provided is equivalent to ERROR_GUID.
- bool **ocrGuidIsEq**(ocrGuid_t guid1, ocrGuid_t guid2)

  Checks if the two GUIDs provided are equivalent.
- bool **ocrGuidIsLt**(ocrGuid_t guid1, ocrGuid_t guid2)

  Checks if guid1 is less than guid2 (useful to implement a total order on GUIDs)

2.5.1. bool ocrGuidIsNull(ocrGuid_t *guid*)

NULL_GUID is a special value representing a NULL ocrGuid_t. This returns true if **guid** is equivalent to NULL_GUID.

Parameters

in	**guid**	GUID to be evaluated.

Returns A boolean:

- true: guid is equivalent to NULL_GUID
- false: guid is not equivalent to NULL_GUID

2.5.2. bool ocrGuidIsUninitialized(ocrGuid_t *guid*)

UNINITIALIZED_GUID is a special value representing an uninitialized ocrGuid_t. This function checks if the GUID provided is equivalent to UNINITIALIZED_GUID.

Parameters

in	**guid**	GUID to be evaluated.

Returns A boolean:

- true: guid is equivalent to UNINITIALIZED_GUID

- false: guid is not equivalent to UNINITIALIZED_GUID

2.5.3. bool ocrGuidIsError(ocrGuid_t *guid*)

ERROR_GUID is a special value representing a invalid ocrGuid_t. This function checks if the GUID provided is equivalent to ERROR_GUID.

Parameters

in	**guid**	GUID to be evaluated.

Returns A boolean:

- true: guid value is equivalent to ERROR_GUID
- false: guid value is not equivalent to ERROR_GUID

2.5.4. bool ocrGuidIsEq(ocrGuid_t *guid1,* ocrGuid_t *guid2*)

This function checks if guid1 and guid2 are equivalent.

Parameters

in	**guid1**	GUID to be evaluated.
in	**guid2**	GUID to be evaluated.

Returns A boolean:

- true: guid1 and guid2 are equivalent
- false: guid1 and guid2 are not equivalent

2.5.5. bool ocrGuidIsLt(ocrGuid_t *guid1,* ocrGuid_t *guid2*)

This function checks if guid1 is less than guid2.

Parameters

in	**guid1**	GUID to be evaluated.
in	**guid2**	GUID to be evaluated.

Returns A boolean:

- true: guid1 is less than guid2

- false: guid1 is greater than or equal to guid2

2.5.6. Macros for printing GUIDs

It is often desirable in both applications and runtime to print the value of a GUID for debugging purposes. Under previous assumptions that GUIDs were **`unsigned long`** values, they were printed using the **`"%PRIx64"`** placeholder. Since GUIDs are opaque, their size cannot be assumed and format specifiers are therefore provided to properly print GUIDs.

- **`GUIDF`** *Format placeholder representing a GUID.*
- **`GUIDA(guid)`** *Expands guid value into format arguments corresponding to the GUIDF format placeholder.*

The following snippet shows how to print a GUID:

```
1 ocrPrintf("DB Guid "GUIDF" acquired by "GUIDF"\n", GUIDA(db.guid), GUIDA(edt.guid));
```

Note that the **`GUIDF`** format placeholder macro may actually expand into multiple concrete format placeholders. Likewise, the **`GUIDA`** macro may expand its single GUID argument into multiple arguments for a format function's argument list, corresponding to the number of concrete placeholders included by the **`GUIDF`** macro.

2.6. Data block management

Data blocks are the only form of non-ephemeral storage and are therefore the only way to "share" data between EDTs. Conceptually, data blocks are contiguous chunks of memory that have a start address and a size. They also have the following characteristics:

- all memory within the data block is accessible from the start-address using an offset (ie: addresses [start-address; start-address+size[uniquely and totally address the entire data-block);
- a data block is non-overlapping with other distinct data blocks
- the pointer to the start of a data block is only valid between the start of the EDT (or the data block creation) and the corresponding **`ocrDbRelease`** (or the end of the EDT).

The following macros and enums are used with OCR data blocks.

- **`enum ocrInDbAllocator_t`** containing:
  - **`NO_ALLOC`** The data block is not used as a heap
- **`enum ocrDbAccessMode_t`** containing the four access modes. The meaning of the access modes is given in Section 1.4.4:
  - **`DB_MODE_RW`**

  - **DB_MODE_EW**
  - **DB_MODE_RO**
  - **DB_MODE_CONST**
  - **DB_MODE_NULL**
- **DB_DEFAULT_MODE** which is an alias of **DB_MODE_RW**
- **DB_PROP_NONE** which specifies no special properties on the data block
- **DB_PROP_NO_ACQUIRE** which specifies that the data block should not be acquired on creation

Functions

- u8 **ocrDbCreate**(ocrGuid_t *db, void **addr, u64 len, u16 flags, const ocrHint_t *hint, ocrInDbAllocator_t allocator)

  Request the creation of a data block.
- u8 **ocrDbDestroy**(ocrGuid_t db)

  Request the destruction of a data block.
- u8 **ocrDbRelease**(ocrGuid_t db)

  Release the data block indicating the end of its use by the EDT.
- u8 **ocrDbDowngradeRelease**(ocrGuid_t db)

  Release updates made to a writable data block while retaining read-only access to the data block.

2.6.1. u8 ocrDbCreate(ocrGuid_t * *db,* void ** *addr,* u64 *len,* u16 *flags,* const ocrHint_t * *hint,* ocrInDbAllocator_t *allocator*)

Requests the creation of a data block of the specified size. After a successful call, the runtime will return the GUID for the newly created data block and a pointer to it (if requested).

Parameters

out	**db**	On successful creation, contains the GUID of the data block. If the call fails, the returned value is undefined.
out	**addr**	On successful creation and if the DB_PROP_NO_ACQUIRE is not specified in **flags**, the created data block will be acquired and its starting address will be returned in this parameter. If DB_PROP_NO_ACQUIRE is specified in **flags**, the value returned will be NULL. If the call fails, the returned value is undefined
in	**len**	Non-zero size, in bytes, of the data block to create

in	**flags**	Flags controlling the behavior of the data block creation. The supported flags are: • DB_PROP_NONE: Default behavior (described in this section) • DB_PROP_NO_ACQUIRE: The created data block may not be used by this EDT (NULL will be returned in **addr**). Note that a conforming implementation may delay the creation of the data block until it is acquired by another EDT.
in	**hint**	Reserved for future use. This parameter should be NULL_HINT
in	**allocator**	A data block can be used as the backing memory for malloc/free-like operations. This parameter specifies the allocator to use for these operations inside this data block. Note that if a data block allocator is used, the user should not write directly to the data block as this may overwrite the meta data used by the allocator. The 'NO_ALLOC' allocator is used to indicate that the data block is not used by any allocator and is therefore entirely usable for user data. Supported values for this parameter are given by ocrInDbAllocator_t.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): The arguments passed (flags, allocator, etc.) were not valid
- 0: Successful
- OCR_ENOMEM (**deferred**): The runtime could not provide a data block of size 'len' due to insufficient memory
- OCR_EBUSY (**deferred**): A resource required for this call was busy. A retry is possible

Description This function is used to create the basic unit of data in OCR: the data block. Unless DB_PROP_NO_ACQUIRE is specified in **flags**, this function also acquires the newly created data block and returns a pointer to the start of the data block in **addr**.

The created data block:

- Will always be 8-byte aligned
- Will not necessarily be zeroed out; the value of the created data block is undefined;
- Will have a GUID that is unique from this call until the user calls **ocrDbDestroy()** on this data block.

Note

- Using the DB_PROP_NO_ACQUIRE flag is recommended to allow the runtime to make

placement decisions for newly created data blocks. Not specifying this flag may result in a sub-optimal memory placement for the created data block.

- When DB_PROP_NO_ACQUIRE is specified, a conformant implementation may choose not to create the data block immediately and instead create it lazily before the using EDT runs.
- Like all GUIDs, the uniqueness of a data block's GUID is not necessarily unique throughout the entire program. A data block's GUID is guaranteed unique only as long as the data block exists (between **`ocrDbCreate()`** and **`ocrDbDestroy()`**).

2.6.2. u8 ocrDbDestroy(ocrGuid_t *db*)

Request for the destruction of a data block. All created data blocks should be destroyed when no longer needed to reclaim the space they utilize.

Parameters

in	**db**	GUID of the data block to destroy

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EPERM (**deferred**): The data block was already destroyed
- OCR_EINVAL (**deferred**): The GUID passed as argument does not refer to a valid data block

Description OCR does not perform automatic garbage collection; all created data blocks therefore need to be explicitly destroyed by the user. This function will request the destruction of a data block but said destruction will be delayed until all EDTs that have acquired the data block have released it (either explicitly with **`ocrDbRelease()`** or by transitioning to the *released* state).

This function does not need to be called by an EDT that has acquired the data block. If the EDT did acquire the data block, however, this function will implicitly release it (equivalent to calling **`ocrDbRelease()`**.

Note

Not all instances of the errors indicated by the error codes can be caught. In other words, attempting the destroy a data block multiple times may result in the return of an error code but may also result in undefined behavior.
At the end of an OCR program, all data blocks are destroyed by the OCR runtime.

2.6.3. u8 ocrDbRelease(ocrGuid_t *db*)

An EDT acquires a data block either on creation with **`ocrDbCreate()`** or implicitly when it transitions to the *ready* state. All acquired data blocks will be implicitly released by the runtime

when the EDT transitions to the *released* state but they can be released earlier using this function. Releasing a data block indicates that the EDT no longer has use for it and also enables other EDTs to "see" the eventual changes to the data block (release consistency).

Parameters

in	**db**	GUID of the data block to release

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): The GUID passed as argument does not refer to a valid data block
- OCR_EACCESS (**immediate**): The calling EDT has not acquired the data block and therefore cannot release it

Description This function is critical in ensuring proper memory ordering in OCR. A data block can be "shared" with another EDT B by satisfying an event that B depends on. B is only guaranteed to see the changes made to the data block by this EDT if this EDT releases the data block before satisfying the event B depends on with this data block.

Note

Once the EDT releases a data block, it can no longer read or write to it (the pointer it had to it should be considered invalid). Violating this rule will result in undefined behavior. A consequence of this is that a data block can only be released at most once by an EDT.

2.6.4. u8 ocrDbDowngradeRelease(ocrGuid_t *db*)

This function allows for any writes made to a data block in the current EDT to be *released* (in terms of OCR's release consistency memory model) while still allowing the data block to be read later in the current EDT. In contrast, a call to `ocrDbRelease()` indicates the end of both write and read access to the target data block within the current EDT.

Parameters

in	**db**	GUID of the data block to downgrade/release

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**deferred**): The GUID passed as argument does not refer to a valid data block
- OCR_EACCESS (**immediate**): The calling EDT has not acquired the data block and therefore cannot downgrade/release it

Description OCR's release-consistency memory model requires a *release* operation on a data block for synchronization, ensuring that writes to that data block are visible to all EDTs ordered after the *release* operation. For a data block that is acquired in a writable mode by the current EDT, this function performs a *release* operation (making all preceding writes visible in terms of the memory model); however, the current EDT retains read-only access to the data block by *downgrading* the data block's access mode to a read-only access mode (i.e., **DB_MODE_RO** or **DB_MODE_CONST**). The user should assume that the mode for the downgraded data block is **DB_MODE_RO** but conformant implementations may downgrade certain data blocks to **DB_MODE_CONST**. If the target data block is already in a read-only access mode, then this function does nothing.

Note

Unlike **ocrDbRelease()**, it is safe to call this function multiple times with the same target data block within a single EDT. The target data block can later be fully released via a call to **ocrDbRelease()**, otherwise it will be released implicitly at the end of the current EDT.

2.7. Event Management

Events are used to coordinate the execution of tasks and to help establish dependences in the directed acyclic graph representing the execution of an OCR program.

The following macros and enums are used with OCR events:

- **enum ocrEventTypes_t** containing the types of supported events. The meaning of these event types is given in Section 1.4.3:
  - **OCR_EVENT_ONCE_T**
  - **OCR_EVENT_IDEM_T**
  - **OCR_EVENT_STICKY_T**
  - **OCR_EVENT_LATCH_T**
- **enum ocrLatchEventSlots_t** containing constants to identify the two pre-slots of the latch event type:
  - **OCR_EVENT_LATCH_DECR_SLOT** identifying the decrement slot of the latch event. The numeric equivalent of this value is 0.
  - **OCR_EVENT_LATCH_INCR_SLOT** identifying the increment slot of the latch event. The numeric equivalent of this value is 1.

Functions

- u8 **ocrEventCreate**(ocrGuid_t *guid, ocrEventTypes_t eventType, u16 flags)

  Request the creation of an event.

- u8 **ocrEventDestroy**(ocrGuid_t guid)

  Explicitly destroys an event.

- u8 **ocrEventSatisfy**(ocrGuid_t eventGuid, ocrGuid_t dataGuid)

  Satisfy the first pre-slot of an event and optionally pass a data block to the event.

- u8 **ocrEventSatisfySlot**(ocrGuid_t eventGuid, ocrGuid_t dataGuid, u32 slot)

  Satisfy the specified pre-slot of an event and optionally pass a data block to the event. Note that, in the case of a latch event, the slot parameter is of type enum ocrLatchEventSlots_t

2.7.1. u8 ocrEventCreate(ocrGuid_t * *guid*, ocrEventTypes_t *eventType*, u16 *flags*)

Requests the creation of an event of the specified type. After a successful call, the runtime will return the GUID for the newly created event. The returned GUID is immediately usable.

Parameters

out	**guid**	On successful creation, contains the GUID of the event. If the call fails, the returned value is undefined.
in	**eventType**	The type of event to create. See .
in	*flags*	Flags impacting the creation of the event. Currently, the following flags are supported: • EVT_PROP_NONE: Default behavior • EVT_PROP_TAKES_ARG: If this flag is **not** set, the event will ignore any data block passed to it when it is satisfied. The satisfaction call will raise a deferred error if a data block is passed in the satisfy call. If the flag is set, the event can take a data block or NULL_GUID when it is satisfied. In other words, EVT_PROP_TAKES_ARG, if set, only tells the runtime that the event **may** take a data block on satisfaction but does not force it to.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): The **eventType** argument is invalid or incompatible with **flags**

- OCR_ENOMEM (**deferred**): The runtime could not create the event due to insufficient memory

Description This function is used to create the basic synchronization mechanism is OCR: the event. The lifetime of the created event is dependent on its type. See Section 1.4.3 for more detail.

2.7.2. u8 ocrEventDestroy(ocrGuid_t *guid*)

Certain event types, specifically *sticky* or *idempotent* events do not get automatically destroyed when they are satisfied. The user must explicitly destroy these events when they are no longer needed.

Parameters

in	**guid**	GUID of the event to destroy.

Returns 0 if no immediate error was detected or the following error codes:

- 0: Successful
- OCR_EINVAL (**deferred**): The GUID passed as argument does not refer to a valid event

Description *Once* and *latch* events are automatically destroyed by the runtime when they trigger and propagate their satisfaction to the objects connected to their post-slots; those events should not be destroyed using this function. Other events, however, need to be destroyed when they are no longer needed.

Note

If, before this call, the event has EDTs waiting on it that are not yet in the *ready* state, those EDTs will never start unless their dependences are reset to another event. If the waiting EDTs are in the *runnable* state, the behavior is undefined.
Using the GUID of an event after it has been destroyed using this call will result in undefined behavior.

2.7.3. u8 ocrEventSatisfy(ocrGuid_t *eventGuid,* ocrGuid_t *dataGuid*)

Equivalent to **`ocrEventSatisfySlot(eventGuid, dataGuid, 0)`**. See Section 2.7.4 for more detail.

Parameters

in	**eventGuid**	GUID of the event to satisfy
in	**dataGuid**	GUID of the data block to pass to the event or NULL_GUID if this event does not take any data blocks (pure control dependence)

Returns 0 if no immediate error was detected or the following error codes:

- OCR_ENOMEM (**deferred**): The runtime could not satisfy the event due to insufficient memory
- OCR_EINVAL (**deferred**): **`eventGuid`** or **`dataGuid`** do not refer to valid event or data block GUIDs respectively
- OCR_EACCES (**deferred**): a non NULL_GUID was passed as **`dataGuid`** when the event does not take an argument
- OCR_EPERM (**deferred**): The event has already been satisfied and does not support multiple satisfactions or a data block was passed to an event which does not take arguments

Description See Section 2.7.4 for a detailed discussion of this function.

2.7.4. u8 ocrEventSatisfySlot(ocrGuid_t *eventGuid,* ocrGuid_t *dataGuid,* u32 *slot*)

Satisfy the specified pre-slot of an event thereby potentially causing waiting EDTs to become *runnable*. This function is the primary method of synchronization in OCR.

Parameters

in	**`eventGuid`**	GUID of the event to satisfy
in	**`dataGuid`**	GUID of the data block to pass to the event or NULL_GUID if this event does not take any data blocks (pure control dependence)
in	**`slot`**	Pre-slot on the destination event to satisfy

Returns 0 if no immediate error was detected or the following error codes:

- OCR_ENOMEM (**deferred**): The runtime could not satisfy the event due to insufficent memory
- OCR_EINVAL (**deferred**): **`eventGuid`** or **`dataGuid`** do not refer to valid event or data block GUIDs respectively
- OCR_EPERM (**deferred**): The event has already been satisfied and does not support multiple satisfactions or a data block was passed to an event which does not take arguments

Description Satisfying the pre-slot of an event will potentially trigger the satisfaction of its post-slot depending on its trigger rule:

- *Once*, *idempotent* and *sticky* events will satisfy their post-slot upon satisfaction of their pre-slot
- *Latch* events will trigger if and only if the number of satisfactions on both their pre-slots is equal. See Section 1.4.3 for more detail.

The **dataGuid** argument is used to associate a data block with the event. *Once*, *idempotent* and *sticky* events will pass this data block down their post-slot. An EDT connected to the post-slot of the event (or the post-slot of a chain of events connected to this event) will acquire the data block associated with this event when it transitions to the *ready* state. A data block passed to a *latch* event is ignored.

Note

OCR's memory model (see Section 1.6) imposes that, to guarantee the visibility of the writes to a data block passed to an event (and therefore potentially immediately acquired by an EDT), data blocks need to be *released* with **ocrDbRelease** prior to the satisfaction of the event. Failure to follow this rule will result in undefined behavior. Note that data blocks written to by a preceding EDT will already have been released when that EDT finished and moved to the *release* stage.

2.8. Task management

Event driven tasks – EDTs – act as the task abstraction in OCR, and all program computations are expressed using EDTs.

A support type for EDTs is the EDT template which factor out some information about EDTs. To create an EDT, an EDT template first needs to be created. The EDT template can be reused for all instances of an EDT of the same type.

The following macros and enums are used with OCR tasks.

- **EDT_PROP_NONE** which specifies no special properties for the creation of an EDT
- **EDT_PROP_FINISH** which specifies that the created EDT is a *finish* EDT
- **EDT_PROP_OEVT_VALID** which specifies that the output event passed when creating an EDT is a valid event
- **EDT_PARAM_UNK** which specifies that the number of parameters or dependences to an EDT template is unknown and each EDT will specify the number of parameters or dependences. This allows a single template to be created for EDTs that can take a varying number of parameters or dependences.

The prototype of an EDT function is given by ocrGuid_t (∗**ocrEdt_t**)(u32 **paramc**, u64 ∗**paramv**, u32 **depc**, ocrEdtDep_t **depv[]**)

Functions

- u8 **ocrEdtTemplateCreate**(ocrGuid_t ∗guid, ocrEdt_t funcPtr, u32 paramc, u32 depc)

  Request the creation of an EDT template.

- u8 **ocrEdtTemplateDestroy**(ocrGuid_t guid)

  Request the destruction of an EDT template.

- u8 **ocrEdtCreate**(ocrGuid_t *guid, ocrGuid_t templateGuid, u32 paramc, const u64 *paramv, u32 depc, const ocrGuid_t *depv, u16 flags, const ocrHint_t *hint, ocrGuid_t *outputEvent)

  Request the creation of an EDT instance using the specified EDT template

- u8 **ocrEdtDestroy**(ocrGuid_t guid)

  Request the explicit destruction of an EDT.

2.8.1. u8 ocrEdtTemplateCreate(ocrGuid_t * *guid,* ocrEdt_t *funcPtr,* u32 *paramc,* u32 *depc*)

The EDT template encapsulates information concerning the basic signature and behavior of all EDTs created based on the template. This function creates an EDT template.

Parameters

out	**guid**	On successful creation, contains the GUID of the EDT template. If the call fails, the returned value is undefined.
in	**funcPtr**	The function the EDT will execute when it runs. This function must be of type **ocrEdt_t**.
in	**paramc**	The number of parameters EDTs created based on this template will take. If EDTs created based on this template can take a variable number of arguments, the constant EDT_PARAM_UNK can be used.
in	**depc**	The number of pre-slots that EDTs created based on this template will take. If EDTs created based on this template can take a variable number of arguments, the constant EDT_PARAM_UNK can be used.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_ENOMEM (**deferred**): The runtime could not allocate the template

Description An EDT template encapsulates the EDT function and, optionally, the number of parameters and arguments that EDTs instantiated from this template will use. It needs to be created only once for each function that will serve as an EDT.

Note

If the runtime is compiled with OCR_ENABLE_EDT_NAMING, the name of the function will also be stored in the EDT template object to aid in debugging.

2.8.2. u8 ocrEdtTemplateDestroy(ocrGuid_t *guid*)

Destroy an EDT template.

Parameters

in	**guid**	GUID of the EDT template to destroy

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**deferred**): The GUID passed as argument does not refer to a valid EDT template

Description This function will destroy the EDT template object.

Note

The destruction of the EDT template can occur even if all EDTs created from it have not run to completion. The EDT template cannot, however, be used to create new EDTs once it has been destroyed.

2.8.3. u8 ocrEdtCreate(ocrGuid_t * *guid*, ocrGuid_t *templateGuid*, u32 *paramc*, const u64 * *paramv*, u32 *depc*, const ocrGuid_t * *depv*, u16 *flags*, const ocrHint_t * *hint*, ocrGuid_t * *outputEvent*)

Creates a new instance of an EDT based on the specified EDT template. In OCR, an EDT will run at most once and be automatically destroyed once it completes execution.

Parameters

out	**guid**	On successful creation, contains the GUID of the EDT. If the call fails, the returned value is undefined.
in	**templateGuid**	GUID of the template to use to create the EDT.
in	**paramc**	Number of parameters (64-bit values) contained in the **paramv** array.
in	**paramv**	Pointer to an array of **paramc** u64 values. The values are copied in and can therefore be freed/reused after this call returns. If **paramc** is 0, this parameter must be set to NULL.
in	**depc**	Number of pre-slots for this EDT.

in	**depv**	Pointer to an array of **depc** ocrGuid_t values or NULL. All pre-slots specified in this way will be acquired using **DB_DEFAULT_MODE**. Use **ocrAddDependence()** to specify an alternate mode. If you only want to specify some of the pre-slots use **UNINITIALIZED_GUID** for the unknown ones.
in	**flags**	Flags controlling the behavior of EDT creation. The supported flags are: • EDT_PROP_NONE: Regular EDT • EDT_PROP_FINISH: Create a *finish* EDT • EDT_PROP_OEVT_VALID: The output event's (last argument) value is a valid event to use as an output event
in	**hint**	Reserved for future use. Set to NULL_HINT.
in,out	**outputEvent**	The meaning of this field is controlled by the value of the 'flags' argument. If EDT_PROP_OEVT_VALID is specified in the flags, outputEvent must be non-NULL and point to the GUID of a valid user created event. Note that if the event given as an argument is a latch event, only the OCR_EVENT_LATCH_DECR_SLOT is satisfied. If EDT_PROP_OEVT_VALID is not specified in the flags: if non-NULL on input, on successful creation, contains the GUID of the event associated with the post-slot of the EDT; if NULL, no output event will be returned. When the event is automatically created by the runtime (if EDT_PROP_OEVT_VALID is not specified), the event returned will be automatically destroyed on satisfaction. It is therefore crucial to ensure that all waiters on this event are set up properly *before* the EDT transitions to the *runnable* state. If this behavior is undesirable, the EDT_PROP_OEVT_VALID allows you to override the event that will be used as the output event and the programmer can provide a sticky event which will not be destroyed on satisfaction. In all cases, the post-slot of the output event (either specified by the user or created by the runtime) will be satisfied: **a)** for a *finish* EDT, when the EDT and all of its descendant EDTs (the closure of all EDTs created in this EDT and its children) have completed execution or **b)** for a non finish EDT, the post-slot of this event will be satisfied when the EDT completes execution and will carry the data block returned by the EDT.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_ENOMEM (**deferred**): The runtime could not create the EDT
- OCR_EINVAL (**deferred**): The GUID specifying the template is invalid or the number of

parameters or pre-slots are invalid or the combination of flags and other parameter values is invalid

Description The EDT is created based on the function provided during the creation of its template. It is required that the number of parameters (**paramc**) and number of pre-slots (**depc**) be resolved at this time. In other words, the number of parameters and dependences must be explicitly specified at this time. If the EDT template used for this EDT specified specific numbers of parameters or dependences, the values for **paramc** and **depc** must match those specified for the template.

Note

If the EDT is created with all its pre-slots specified and resolved, it may execute immediately, which means that the GUIDs returned (**guid** and **outputEvent**, if created by the runtime) may be invalid by the time this function returns. It is the responsibility of the programmer – if these returned values are needed – to ensure that the EDT cannot start until a later time (e.g., by inserting an extra pre-slot that is satisfied only after setting up all dependences on the output event).

2.8.4. u8 ocrEdtDestroy(ocrGuid_t *guid*)

EDTs are automatically destroyed after they execute. This call provides a way to explicitly destroy a created EDT if the programmer later realizes that it will never become *runnable*.

Parameters

in	**guid**	GUID of the EDT to destroy

Description Most programmers will not have use for this function as OCR implicitly destroys all EDTs that execute. There are some cases, however, where an EDT is created and the programmer later realizes that the task will never execute. This function allows the programmer to reclaim the memory used by the EDT. If the destroyed EDT is a descendant of a finish EDT, the destruction preserves the finish EDT's property – i.e., its latch event is updated so that the finish EDT is allowed to trigger its post-slot as if the destroyed EDT was not part of its finish scope.

Note

Destroying an EDT that is in the *runnable* state or later will result in undefined behavior. If the EDT had an associated **outputEvent**, that event will also be explicitly destroyed.

Returns 0 if no immediate error was detected. There are no error codes for this function.

2.9. Dependence management

At the core of OCR is the concept of a directed acyclic graph that represents the evolving state of an OCR program. EDTs become runnable once their dependences are met. These dependences can be set explicitly when creating the EDT or dynamically using the functions from this section of the API.

Functions

- u8 **ocrAddDependence**(ocrGuid_t source, ocrGuid_t destination, u32 slot, ocrDbAccessMode_t mode)

  Adds a dependence between OCR objects

2.9.1. u8 ocrAddDependence(ocrGuid_t *source*, ocrGuid_t *destination*, u32 *slot*, ocrDbAccessMode_t *mode*)

Adds a dependence between two OCR objects. Concretely, this will link the post-slot of the **source** object to the **slot**th pre-slot of **destination**. When a dependence exists between post-slot *A* and pre-slot *B*, when *A* becomes satisfied, *B* will also become satisfied.

Parameters

in	**source**	GUID of the source OCR object. The source can be a data block or an event or NULL_GUID
in	**destination**	GUID of the destination OCR object. The destination can be an event or an EDT
in	**slot**	Index of the pre-slot on **destination**
in	**mode**	If **destination** is an EDT, access mode with which the EDT will access the data block. If **destination** is an event, this value is ignored.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**deferred**): **slot** is invalid
- OCR_ENOPERM (**deferred**): **source** and **destination** cannot be linked with a dependence (for example, if **destination** is a data block)

Description The following dependences can be added:

- Event to event: The destination event's pre-slot will become satisfied upon satisfaction of the source event's post-slot. Any data block associated with the source event's post-slot will be

associated with the sink event's pre-slot, except when the sink event is a latch event. This allows the formation of event chains.

- Event to EDT: Upon satisfaction of the source event's post-slot, the EDT's pre-slot will be satisfied. When the runtime transitions the EDT from the *runnable* to *ready* state, the data block associated with the post-slot of the event will be acquired using the **mode** specified.
- Data block to event: Adding a dependence between a data block and an event will result in the eventual satisfaction of the event with the data block; i.e., **ocrAddDependence(db, evt, slot, mode)** is similar to **ocrEventSatisfySlot(evt, db, slot)**, but the satisfaction of the target event is not guaranteed to happen synchronously.
- Data block to EDT: This represents a pure data dependence. Adding a dependence between a data block and an EDT immediately satisfies the pre-slot of the EDT. When the runtime transitions the EDT from the *runnable* to the *ready* state, the data block will be acquired using the **mode** specified.
- NULL_GUID to event or EDT: This is equivalent to "data block to event" or "data block to EDT" where the data is non-existent.

Note that an EDT instance may legally have multiple pre-slots satisfied with the same data block if those dependences have the same access mode. An EDT instance acquiring the same data block through multiple pre-slots, but with differing access modes, results in undefined behavior. The user is always responsible for ensuring that each data block is explicitly released (via **ocrDbRelease**) at most once per EDT instance.

Bibliography

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. 2

[2] S. Chatterjee. *Runtime Systems for Extreme Scale Platforms*. PhD thesis, Rice University, Dec 2013. 13

[3] S. Chatterjee, S. Tasirlar, Z. Budimlić, V. Cavé, M. Chabbi, M. Grossman, Y. Yan, and V. Sarkar. Integrating Asynchronous Task Parallelism with MPI. In *IPDPS '13: Proceedings of the 2013 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, 2013. 1, 16

[4] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM. 2

[5] Y. Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010. 1, 16

[6] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, May 2009. IEEE Computer Society. 13

[7] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, Apr 2010. IEEE Computer Society. 1, 16

[8] S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *28th European Conference on Object-Oriented Programming (ECOOP)*, Jul 2014. 9

[9] V. Sarkar et al. DARPA Exascale Software Study report, September 2009. 2

[10] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale. 2

[11] D. Sbîrlea, Z. Budimlić, and V. Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 343–356, New York, NY, USA, 2014. ACM. 1, 16

[12] D. Sbîrlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 601–613, Berlin, Heidelberg, 2012. Springer-Verlag. 6

[13] D. Sbîrlea, A. Sbîrlea, K. B. Wheeler, and V. Sarkar. The Flexible Preconditions Model for Macro-Dataflow Execution. In *The 3rd Data-Flow Execution Models for Extreme Scale Computing Workshop (DFM)*, Sep 2013. 1

[14] J. Shirako, V. Cave, J. Zhao, and V. Sarkar. Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism. In *The 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, March 2013. 2

[15] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011. 1, 12

[16] S. Tasirlar. Scheduling Macro-Dataflow Programs on Task-Parallel Runtime Systems, Apr 2011. 1, 12

[17] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar. A practical approach to doacross parallelization. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 219–231, Berlin, Heidelberg, 2012. Springer-Verlag. 6

[18] N. Vrvilo. Asynchronous Checkpoint/Restart for the Concurrent Collections Model, Aug 2014. MS thesis. 1

[19] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM. 1, 12

A. OCR Examples

This chapter demonstrates the use of OCR through a series of examples. The examples are ordered from the most basic to the most complicated and frequently make use of previous examples. They are meant to guide the reader in understanding the fundamental concepts of the OCR programming model and API.

A.1. OCR's "Hello World!"

This example illustrates the most basic OCR program: a single function that prints the message "Hello World!" on the screen and exits.

A.1.1. Code example

The following code will print the string "Hello World!" to the standard output and exit. Note that this program is fully functional (ie: there is no need for a `main` function).

```
#include <ocr.h>

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrPrintf('Hello World!\n');
    ocrShutdown();
    return NULL_GUID;
}
```

A.1.1.1. Details

The `ocr.h` file included on Line 1 contains all of the main OCR APIs. Other more experimental or extended APIs are also located in the `extensions/` folder of the include directory.

EDT's signature is shown on Line 3. A special EDT, named `mainEdt` is called by the runtime if the programmer does not provide a `main` function[1].

[1] Note that if the programmer *does* provide a `main` function, it is the responsability of the programmer to properly initialize the runtime, call the first EDT to execute and properly shutdown the runtime. Refer to the legacy mode extension and the `ocr-legacy.h` header file for more detail.

The `ocrShutdown` function called on Line 5 should be called once and only once by all OCR programs to indicate that the program has terminated. The runtime will then shutdown and any non-executed EDTs at that time are not guaranteed to execute.

A.2. Expressing a fork-join pattern

This example illustrates the creation of a fork-join pattern in OCR.

A.2.1. Code example

```
/* Example of a "fork-join" pattern in OCR
 *
 * Implements the following dependence graph:
 *
 *   mainEdt
 *   /     \
 * fun1    fun2
 *   \     /
 * shutdownEdt
 *
 */

#include "ocr.h"

ocrGuid_t fun1(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid,(void **) &k, sizeof(int), 0, NULL_HINT, NO_ALLOC);
    k[0]=1;
    ocrPrintf("Hello from fun1, sending k = %"PRId32"\n",*k);
    return db_guid;
}

ocrGuid_t fun2(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid,(void **) &k, sizeof(int), 0, NULL_HINT, NO_ALLOC);
    k[0]=2;
    ocrPrintf("Hello from fun2, sending k = %"PRId32"\n",*k);
    return db_guid;
}

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrPrintf("Hello from shutdownEdt\n");
    int* data1 = (int*) depv[0].ptr;
    int* data2 = (int*) depv[1].ptr;
    ocrPrintf("Received data1 = %"PRId32", data2 = %"PRId32"\n", *data1, *data2);
    ocrDbDestroy(depv[0].guid);
    ocrDbDestroy(depv[1].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrPrintf("Starting mainEdt\n");
    ocrGuid_t edt1_template, edt2_template, edt3_template;
```

```
    ocrGuid_t edt1, edt2, edt3, outputEvent1, outputEvent2;
48
    //Create templates for the EDTs
    ocrEdtTemplateCreate(&edt1_template, fun1, 0, 1);
    ocrEdtTemplateCreate(&edt2_template, fun2, 0, 1);
    ocrEdtTemplateCreate(&edt3_template, shutdownEdt, 0, 2);
53
    //Create the EDTs
    ocrEdtCreate(&edt1, edt1_template, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_HINT, &outputEvent1);
    ocrEdtCreate(&edt2, edt2_template, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_HINT, &outputEvent2);
    ocrEdtCreate(&edt3, edt3_template, EDT_PARAM_DEF, NULL, 2, NULL, EDT_PROP_NONE,
        NULL_HINT, NULL);
58
    //Setup dependences for the shutdown EDT
    ocrAddDependence(outputEvent1, edt3, 0, DB_MODE_CONST);
    ocrAddDependence(outputEvent2, edt3, 1, DB_MODE_CONST);

63  //Start execution of the parallel EDTs
    ocrAddDependence(NULL_GUID, edt1, 0, DB_DEFAULT_MODE);
    ocrAddDependence(NULL_GUID, edt2, 0, DB_DEFAULT_MODE);
    return NULL_GUID;
}
```

A.2.1.1. Details

The `ocr.h` file included on Line 13 contains all of the main OCR APIs. The `mainEdt` is shown on Line 44. It is called by the runtime as a `main` function is not provided (more details in `hello.c`).

The `mainEdt` creates three templates (Lines 50, 51 and 52), respectively for three different EDTs (Lines 55, 56 and 57). An EDT is created as an instance of an EDT template. This template stores metadata about the EDTs created from it, optionally defines the number of dependences and parameters used when creating an instance of an EDT, and is a container for the function that will be executed by an EDT (called the EDT function). For the EDTs `edt1`, `edt2` and `edt3`, the EDT functions are, respectively, `fun1`, `fun2` and `shutdownEdt`. The last parameter to `ocrEdtTemplateCreate` is the total number of data blocks on which the EDTs depends. The signature of EDT creation API, `ocrEdtCreate`, is shown in Lines 55, 56 and 57. When `edt1` and `edt2` will complete, they will satisfy the output events `outputEvent1` and `outputEvent2` repectively. This is not required for `edt3`. However, `edt3` should execute only when the events `outputEvent1` and `outputEvent2` are satisfied. This is done by setting up dependences on `edt3` by using the API `ocrAddDependence`, as shown in Lines 60 and 61. This spawns `edt3` but it will not execute until both the events are satisfied. Finally, the EDTs `edt1` and `edt2` are spawned in Lines 64 and 65 respectively. As they do not have any dependences, they execute the associated EDT functions in parallel. These functions (`fun1` and `fun2`) create data blocks using the API `ocrDbCreate` (Lines 18 and 27). The data is written to the data blocks and the GUID is returned (Lines 21 and 30). This will satisfy the events on which the `edt3` is waiting. The EDT function `shutdownEdt` executes and calls `ocrShutdown` after reading and destroying the two data blocks.

A.3. Expressing unstructured parallelism

A.3.1. Code example

This example illustrates several aspect of the OCR API with regards to the creation of an irregular task graph. Specifically, it illustrates:

1. Adding dependences between **a)** events and EDTs, **b)** data blocks and EDTs, and **c)** the NULL_GUID and EDTs;
2. The use of an EDT's post-slot and how a "producer" EDT can pass a data block to a "consumer" EDT using this post-slot;
3. Several methods of satisfying an EDT's pre-slot: **a)** through the use of an explicit dependence array at creation time, **b)** through the use of another EDT's post-slot and **c)** through the use of an explicitly added dependence followed by an `ocrEventSatisfy` call.

```
/* Example of a pattern that highlights the
 * expressiveness of task dependences
 *
 * Implements the following dependence graph:
 *
 * mainEdt
 * |      \
 * stage1a stage1b
 * |     \       |
 * |      \      |
 * |       \     |
 * stage2a  stage2b
 *     \      /
 *     shutdownEdt
 */
#include "ocr.h"

#define NB_ELEM_DB 20

typedef struct {
    ocrGuid_t evtGuid;
} guidPRM_t;

// How many parameters does it take to encode a GUID
#define PARAM_SIZE (sizeof(guidPRM_t) + sizeof(u64) - 1)/sizeof(u64)

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrAssert(depc == 2);
    u64* data0 = (u64*)depv[0].ptr;
    u64* data1 = (u64*)depv[1].ptr;

    ocrAssert(*data0 == 3ULL);
    ocrAssert(*data1 == 4ULL);
    ocrPrintf("Got a DB (GUID "GUIDF") containing %"PRIu64" on slot 0\n",
        GUIDA(depv[0].guid), *data0);
    ocrPrintf("Got a DB (GUID "GUIDF") containing %"PRIu64" on slot 1\n",
        GUIDA(depv[1].guid), *data1);

    // Free the data blocks that were passed in
    ocrDbDestroy(depv[0].guid);
    ocrDbDestroy(depv[1].guid);
```

```
    // Shutdown the runtime
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t stage2a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]);

ocrGuid_t stage1a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrAssert(depc == 1);
    ocrAssert(paramc == PARAM_SIZE);
    // paramv contains the event that the child EDT has to satisfy
    // when it is done

    // We create a data block for one u64 and put data in it
    ocrGuid_t dbGuid = NULL_GUID, stage2aTemplateGuid = NULL_GUID,
        stage2aEdtGuid = NULL_GUID;
    u64* dbPtr = NULL;
    ocrDbCreate(&dbGuid, (void**)&dbPtr, sizeof(u64), 0, NULL_HINT, NO_ALLOC);
    *dbPtr = 1ULL;

    // Create an EDT and pass it the data block we just created
    // The EDT is immediately ready to execute
    ocrEdtTemplateCreate(&stage2aTemplateGuid, stage2a, PARAM_SIZE, 1);
    ocrEdtCreate(&stage2aEdtGuid, stage2aTemplateGuid, EDT_PARAM_DEF,
                 paramv, EDT_PARAM_DEF, &dbGuid, EDT_PROP_NONE, NULL_HINT, NULL);

    // Pass the same data block created to stage2b (links setup in mainEdt)
    return dbGuid;
}

ocrGuid_t stage1b(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrAssert(depc == 1);
    ocrAssert(paramc == 0);

    // We create a data block for one u64 and put data in it
    ocrGuid_t dbGuid = NULL_GUID;
    u64* dbPtr = NULL;
    ocrDbCreate(&dbGuid, (void**)&dbPtr, sizeof(u64), 0, NULL_HINT, NO_ALLOC);
    *dbPtr = 2ULL;

    // Pass the created data block created to stage2b (links setup in mainEdt)
    return dbGuid;
}

ocrGuid_t stage2a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrAssert(depc == 1);
    ocrAssert(paramc == PARAM_SIZE);

    guidPRM_t *params = (guidPRM_t*)paramv;

    u64 *dbPtr = (u64*)depv[0].ptr;
    ocrAssert(*dbPtr == 1ULL); // We got this from stage1a

    *dbPtr = 3ULL; // Update the value

    // Pass the modified data block to shutdown
    ocrEventSatisfy(params->evtGuid, depv[0].guid);

    return NULL_GUID;
}

ocrGuid_t stage2b(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrAssert(depc == 2);
    ocrAssert(paramc == 0);
```

```
    u64 *dbPtr = (u64*)depv[1].ptr;
    // Here, we can run concurrently to stage2a which modifies the value
    // we see in depv[0].ptr. We should see either 1ULL or 3ULL

    // On depv[1], we get the value from stage1b and it should be 2
    ocrAssert(*dbPtr == 2ULL); // We got this from stage2a

    *dbPtr = 4ULL; // Update the value

    return depv[1].guid; // Pass this to the shudown EDT
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {

    ocrGuid_t stage1aTemplateGuid = NULL_GUID, stage1bTemplateGuid = NULL_GUID,
        stage2aTemplateGuid = NULL_GUID, stage2bTemplateGuid = NULL_GUID,
        shutdownEdtTemplateGuid = NULL_GUID;
    ocrGuid_t shutdownEdtGuid = NULL_GUID, stage1aEdtGuid = NULL_GUID,
        stage1bEdtGuid = NULL_GUID, stage2bEdtGuid = NULL_GUID,
        evtGuid = NULL_GUID, stage1aOut = NULL_GUID, stage1bOut = NULL_GUID,
        stage2bOut = NULL_GUID;

    // Create the shutdown EDT
    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0, 2);
    ocrEdtCreate(&shutdownEdtGuid, shutdownEdtTemplateGuid, 0, NULL, EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_HINT, NULL);

    // Create the event to satisfy shutdownEdt by stage 2a
    // (stage 2a is created by 1a)
    ocrEventCreate(&evtGuid, OCR_EVENT_ONCE_T, true);

    guidPRM_t tmp;
    tmp.evtGuid = evtGuid;
    // Create stages 1a, 1b and 2b
    // For 1a and 1b, add a "fake" dependence to avoid races between
    // setting up the event links and running the EDT
    ocrEdtTemplateCreate(&stage1aTemplateGuid, stage1a, PARAM_SIZE, 1);
    ocrEdtCreate(&stage1aEdtGuid, stage1aTemplateGuid, EDT_PARAM_DEF, (u64*)(&tmp),
                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_HINT, &stage1aOut);

    ocrEdtTemplateCreate(&stage1bTemplateGuid, stage1b, 0, 1);
    ocrEdtCreate(&stage1bEdtGuid, stage1bTemplateGuid, EDT_PARAM_DEF, NULL,
                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_HINT, &stage1bOut);

    ocrEdtTemplateCreate(&stage2bTemplateGuid, stage2b, 0, 2);
    ocrEdtCreate(&stage2bEdtGuid, stage2bTemplateGuid, EDT_PARAM_DEF, NULL,
                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_HINT, &stage2bOut);

    // Set up all the links
    // 1a -> 2b
    ocrAddDependence(stage1aOut, stage2bEdtGuid, 0, DB_DEFAULT_MODE);

    // 1b -> 2b
    ocrAddDependence(stage1bOut, stage2bEdtGuid, 1, DB_DEFAULT_MODE);

    // Event satisfied by 2a -> shutdown
    ocrAddDependence(evtGuid, shutdownEdtGuid, 0, DB_DEFAULT_MODE);
    // 2b -> shutdown
    ocrAddDependence(stage2bOut, shutdownEdtGuid, 1, DB_DEFAULT_MODE);

    // Start 1a and 1b
    ocrAddDependence(NULL_GUID, stage1aEdtGuid, 0, DB_DEFAULT_MODE);
    ocrAddDependence(NULL_GUID, stage1bEdtGuid, 0, DB_DEFAULT_MODE);
```

```
    return NULL_GUID;
}
```

A.3.1.1. Details

The snippet of code shows one possible way to construct the irregular task-graph shown starting on Line 5. `mainEdt` will create **a)** `stage1a` and `stage1b` as they are the next things that need to execute but also **b)** `stage2b` and `shutdownEdt` because they are the immediate dominators of those EDTs. In general, it is easiest to create an EDT in its immediate dominator because that allows any other EDTs who need to feed it information (necessarily between its dominator and the EDT in question) to be able to know the value of the opaque GUID created for the EDT. `stage2a`, on the other hand, can be created by `stage1a` as no-one else needs to feed information to it.

Most of the "edges" in the dependence graph are also created in `mainEdt` starting at Line 157. These are either between the post-slot (output event) of a source EDT and an EDT or between a regular event and an EDT. Note also the use of NULL_GUID as a source for two dependences starting at Line 168. A NULL_GUID as a source for a dependence immediately satisfies the destination slot; in this case, it satisfies the unique dependence of `stage1a` and `stage1b` and makes them runable. These two dependences do not exist in the graph shown starting at Line 5 but are crucial to avoid a potential race in the program: the output events of EDTs are similar to ONCE events in the sense that they will disappear once they are satisfied and therefore, any dependence on them must be properly setup prior to their potential satisfaction. In other words, the `ocrAddDependence` calls starting at Line 157 must *happen-before* the satisfaction of `stage1aOut` and `stage1bOut`. This example shows three methods of satisfying an EDT's pre-slots:

- Through the use of an explicit dependence array known at EDT creation time as shown on Line 64;
- Through an output event as shown on Line 68. The GUID passed as a return value of the EDT function will be passed to the EDT's output event (in this case `stage1aOut`). If the GUID is a data block's GUID, the output event will be satisfied with that data block. If it is an event's GUID, the two events will become linked;
- Through an explicit satisfaction as shown on Line 97).

A.4. Using a Finish EDT

A.4.1. Code example

The following code demonstrates the use of Finish EDTs by performing a Fast Fourier Transform on a sparse array of length 256 bytes. For the sake of simplicity, the array contents and sizes are hardcoded, however, the code can be used as a starting point for adding more functionality.

```c
/* Example usage of Finish EDT in FFT.
 *
 * Implements the following dependence graph:
 *
 * MainEdt
 *    |
 *
 * FinishEdt
 * {
 *       DFT
 *      /   \
 * FFT-odd FFT-even
 *      \   /
 *     Twiddle
 * }
 *    |
 * Shutdown
 *
 */

#include "ocr.h"
#include "math.h"

#define N            256
#define BLOCK_SIZE 16

// The below function performs a twiddle operation on an array x_in
// and places the results in X_real & X_imag. The other arguments
// size and step refer to the size of the array x_in and the offset therein
void ditfft2(double *X_real, double *X_imag, double *x_in, u32 size, u32 step) {
    if(size == 1) {
        X_real[0] = x_in[0];
        X_imag[0] = 0;
    } else {
        ditfft2(X_real, X_imag, x_in, size/2, 2 * step);
        ditfft2(X_real+size/2, X_imag+size/2, x_in+step, size/2, 2 * step);
        u32 k;
        for(k=0;k<size/2;k++) {
            double t_real = X_real[k];
            double t_imag = X_imag[k];
            double twiddle_real = cos(-2 * M_PI * k / size);
            double twiddle_imag = sin(-2 * M_PI * k / size);
            double xr = X_real[k+size/2];
            double xi = X_imag[k+size/2];

            // (a+bi)(c+di) = (ac - bd) + (bc + ad)i
            X_real[k] = t_real +
                (twiddle_real*xr - twiddle_imag*xi);
            X_imag[k] = t_imag +

                (twiddle_imag*xr + twiddle_real*xi);
            X_real[k+size/2] = t_real -
                (twiddle_real*xr - twiddle_imag*xi);
            X_imag[k+size/2] = t_imag -
                (twiddle_imag*xr + twiddle_real*xi);
        }
    }
}

// The below function splits the given array into odd & even portions and
// calls itself recursively via child EDTs that operate on each of the portions,
// till the array operated upon is of size BLOCK_SIZE, a pre-defined
// parameter. It then trivially computes the FFT of this array, then spawns
// twiddle EDTs to combine the results of the children.
```

```
ocrGuid_t fftComputeEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t computeGuid = paramv[0];
    ocrGuid_t twiddleGuid = paramv[1];
    double *data = (double*)depv[0].ptr;
    ocrGuid_t dataGuid = depv[0].guid;
    u64 size = paramv[2];
    u64 step = paramv[3];
    u64 offset = paramv[4];
    u64 step_offset = paramv[5];
    u64 blockSize = paramv[6];
    double *x_in = (double*)data;
    double *X_real = (double*)(data+offset + size*step);
    double *X_imag = (double*)(data+offset + 2*size*step);

    if(size <= blockSize) {
        ditfft2(X_real, X_imag, x_in+step_offset, size, step);
    } else {
        // DFT even side
        u64 childParamv[7] = { computeGuid, twiddleGuid, size/2, 2 * step,
                               0 + offset, step_offset, blockSize };
        u64 childParamv2[7] = { computeGuid, twiddleGuid, size/2, 2 * step,
                                size/2 + offset, step_offset + step, blockSize };

        ocrGuid_t edtGuid, edtGuid2, twiddleEdtGuid, finishEventGuid, finishEventGuid2;

        ocrEdtCreate(&edtGuid, computeGuid, EDT_PARAM_DEF, childParamv,
                     EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_HINT,
                     &finishEventGuid);
        ocrEdtCreate(&edtGuid2, computeGuid, EDT_PARAM_DEF, childParamv2,
                     EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_HINT,
                     &finishEventGuid2);

        ocrGuid_t twiddleDependencies[3] = { dataGuid, finishEventGuid, finishEventGuid2 };
        ocrEdtCreate(&twiddleEdtGuid, twiddleGuid, EDT_PARAM_DEF, paramv, 3,
                     twiddleDependencies, EDT_PROP_FINISH, NULL_HINT, NULL);

        ocrAddDependence(dataGuid, edtGuid, 0, DB_MODE_RW);
        ocrAddDependence(dataGuid, edtGuid2, 0, DB_MODE_RW);
    }

    return NULL_GUID;
}

// The below function performs the twiddle operation
ocrGuid_t fftTwiddleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    double *data = (double*)depv[0].ptr;
    u64 size = paramv[2];
    u64 step = paramv[3];
    u64 offset = paramv[4];
    double *x_in = (double*)data+offset;
    double *X_real = (double*)(data+offset + size*step);
    double *X_imag = (double*)(data+offset + 2*size*step);

    ditfft2(X_real, X_imag, x_in, size, step);

    return NULL_GUID;
}

ocrGuid_t endEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t dataGuid = paramv[0];

    ocrDbDestroy(dataGuid);
    ocrShutdown();
    return NULL_GUID;
}
```

```
ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {

    ocrGuid_t computeTempGuid, twiddleTempGuid, endTempGuid;
    ocrEdtTemplateCreate(&computeTempGuid, &fftComputeEdt, 7, 1);
    ocrEdtTemplateCreate(&twiddleTempGuid, &fftTwiddleEdt, 7, 3);
    ocrEdtTemplateCreate(&endTempGuid, &endEdt, 1, 1);
    u32 i;
    double *x;

    ocrGuid_t dataGuid;
    ocrDbCreate(&dataGuid, (void **) &x, sizeof(double) * N * 3, DB_PROP_NONE, NULL_HINT,
        NO_ALLOC);

    // Cook up some arbitrary data
    for(i=0;i<N;i++) {
        x[i] = 0;
    }
    x[0] = 1;

    u64 edtParamv[7] = { computeTempGuid, twiddleTempGuid, N, 1, 0, 0, BLOCK_SIZE };
    ocrGuid_t edtGuid, eventGuid, endGuid;

    // Launch compute EDT
    ocrEdtCreate(&edtGuid, computeTempGuid, EDT_PARAM_DEF, edtParamv,
                 EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_HINT,
                 &eventGuid);

    // Launch finish EDT
    ocrEdtCreate(&endGuid, endTempGuid, EDT_PARAM_DEF, &dataGuid,
                 EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_HINT,
                 NULL);

    ocrAddDependence(dataGuid, edtGuid, 0, DB_MODE_RW);
    ocrAddDependence(eventGuid, endGuid, 0, DB_MODE_RW);

    return NULL_GUID;
}
```

A.4.1.1. Details

The above code contains a total of 5 functions - a `mainEdt()` required of all OCR programs, a `ditfft2()` that acts as the core of the recursive FFT computation, calling itself on smaller sizes of the array provided to it, and three other EDTs that are managed by OCR. They include - `fftComputeEdt()` in Line 65 that breaks down the FFT operation on an array into two FFT operations on the two halves of the array (by spawning two other EDTs of the same template), as well as an instance of `fftTwiddleEdt()` shown in Line 109 that combines the results from the two spawned EDTs by applying the FFT "twiddle" operation on the real and imaginary portions of the array. The `fftComputeEdt()` function stops spawning EDTs once the size of the array it operates on drops below a pre-defined `BLOCK_SIZE` value. This sets up a recursive cascade of EDTs operating on gradually smaller data sizes till the `BLOCK_SIZE` value is reached, at which point the FFT value is directly computed, followed by a series of twiddle operations on gradualy larger data sizes till the entire array has undergone the operation. When this is available, a final EDT termed `endEdt()` in Line 123 is called to optionally output the value of the computed FFT,

and terminate the program by calling `ocrShutdown()`. All the FFT operations are performed on a single data block created in Line 141. This shortcut is taken for the sake of didactic simplicity. While this is programmatically correct, a user who desires reducing contention on the single array may want to break down the data block into smaller units for each of the EDTs to operate upon.

For this program to execute correctly, it is apparent that each of the `fftTwiddleEdt` instances can not start until all its previous instances have completed execution. Further, for the sake of program simplicity, an instance of `fftComputeEdt-fftTwiddleEdt` pair cannot return until the EDTs that they spawn have completed execution. The above dependences are enforced using the concept of *Finish EDTs*. As stated before, a Finish EDT does not return until all the EDTs spawned by it have completed execution. This simplifies programming, and does not consume computing resources since a Finish EDT that is not running, is removed from any computing resources it has used. In this program, no instance of `fftComputeEdt` or `fftTwiddleEdt` returns before the corresponding EDTs that operates on smaller data sizes have returned, as illustrated in Lines 92,95 and 99. Finally, the single `endEdt()` instance in Line 155 is called only after all the EDTs spawned by the parent `fftComputeEdt()` in Line 160, return.

A.5. Accessing a data block with "Read-Write" Mode

This example illustrates the usage model for data blocks accessed with the `Read-Write (RW)` mode. The `RW` mode ensures that only one master copy of the data block exists at any time inside a shared address space. Parallel EDTs can concurrently access a data block under this mode if they execute inside the same address space. It is the programmer's responsibility to avoid data races. For example, two parallel EDTs can concurrently update separate memory regions of the same data block with the `RW` mode.

A.5.1. Code example

```
/* Example usage of RW (Read-Write)
 * data block access mode in OCR
 *
 * Implements the following dependence graph:
 *
 *      mainEdt
 *      [ DB ]
 *       /  \
 * (RW)/     \(RW)
 *    /       \
 * EDT1       EDT2
 *    \       /
 *      [ DB ]
 *   shutdownEdt
 *
 */

#include "ocr.h"

#define N 1000
```

```
ocrGuid_t exampleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 i, lb, ub;
    lb = paramv[0];
    ub = paramv[1];
    u32 *dbPtr = (u32*)depv[0].ptr;

    for (i = lb; i < ub; i++)
        dbPtr[i] += i;

    return NULL_GUID;
}

ocrGuid_t awaitingEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 i;
    ocrPrintf("Done!\n");
    u32 *dbPtr = (u32*)depv[0].ptr;
    for (i = 0; i < N; i++) {
        if (dbPtr[i] != i * 2)
            break;
    }

    if (i == N) {
        ocrPrintf("Passed Verification\n");
    } else {
        ocrPrintf("!!! FAILED !!! Verification\n");
    }

    ocrDbDestroy(depv[0].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u32 i;

    // CHECKER DB
    u32* ptr;
    ocrGuid_t dbGuid;
    ocrDbCreate(&dbGuid, (void**)&ptr, N * sizeof(u32), DB_PROP_NONE, NULL_HINT, NO_ALLOC);
    for(i = 0; i < N; i++)
        ptr[i] = i;
    ocrDbRelease(dbGuid);

    // EDT Template
    ocrGuid_t exampleTemplGuid, exampleEdtGuid1, exampleEdtGuid2, exampleEventGuid1,
        exampleEventGuid2;
    ocrEdtTemplateCreate(&exampleTemplGuid, exampleEdt, 2 /*paramc*/, 1 /*depc*/);
    u64 args[2];

    // EDT1
    args[0] = 0;
    args[1] = N/2;
    ocrEdtCreate(&exampleEdtGuid1, exampleTemplGuid, EDT_PARAM_DEF, args, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_HINT, &exampleEventGuid1);

    // EDT2
    args[0] = N/2;
    args[1] = N;
    ocrEdtCreate(&exampleEdtGuid2, exampleTemplGuid, EDT_PARAM_DEF, args, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_HINT, &exampleEventGuid2);

    // AWAIT EDT
    ocrGuid_t awaitingTemplGuid, awaitingEdtGuid;
    ocrEdtTemplateCreate(&awaitingTemplGuid, awaitingEdt, 0 /*paramc*/, 3 /*depc*/);
```

```
    ocrEdtCreate(&awaitingEdtGuid, awaitingTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
        NULL,
        EDT_PROP_NONE, NULL_HINT, NULL);
    ocrAddDependence(dbGuid,              awaitingEdtGuid, 0, DB_MODE_CONST);
    ocrAddDependence(exampleEventGuid1, awaitingEdtGuid, 1, DB_DEFAULT_MODE);
    ocrAddDependence(exampleEventGuid2, awaitingEdtGuid, 2, DB_DEFAULT_MODE);

    // START
    ocrPrintf("Start!\n");
    ocrAddDependence(dbGuid, exampleEdtGuid1, 0, DB_MODE_RW);
    ocrAddDependence(dbGuid, exampleEdtGuid2, 0, DB_MODE_RW);

    return NULL_GUID;
}
```

A.5.1.1. Details

The mainEdt creates a data block (`dbGuid`) that may be concurrently updated by two children EDTs (`exampleEdtGuid1` and `exampleEdtGuid2`) using the `RW` mode. `exampleEdtGuid1` and `exampleEdtGuid2` are each created with one dependence, while after execution, each of them will satisfy an output event (`exampleEventGuid1` and `exampleEventGuid2`). The satisfaction of these output events will trigger the execution of an awaiting EDT (`awaitingEdtGuid`) that will verify the correctness of the computation performed by the concurrent EDTs. `awaitingEdtGuid` has three input dependences: `dbGuid` is passed into the first input, while the other two would be satisfied by `exampleEventGuid1` and `exampleEventGuid2`. Once `awaitingEdtGuid`'s dependences have been setup, the `mainEdt` satisfies the dependences on `exampleEdtGuid1` and `exampleEdtGuid2` with the data block `dbGuid`.

Both `exampleEdtGuid1` and `exampleEdtGuid2` execute the task function called *exampleEdt*. This function accesses the contents of the data block passed in through the dependence slot `0`. Based on the parameters passed in, the function updates a range of values on that data block. After the data block has been updated, the EDT returns and in turn satisfies the output event. Once both EDTs have executed and satisfied their ouput events, the `awaitingEdtGuid` executes function *awaitingEdt*. This function verifies if the updates done on the data block by the concurrent EDTs are correct. Finally, it prints the result of its verification and calls `ocrShutdown`.

A.6. Accessing a data block with "Exclusive-Write" Mode

The `Exclusive-Write (EW)` mode allows for an easy implementation of mutual exclusion of EDTs. When an EDT depends on one or several data blocks in `EW` mode, the runtime guarantees that only one EDT in the entire system will have write access to the data block Hence, the `EW` mode is useful when one wants to guarantee there is no race condition writing to a data block or when ordering among EDTs does not matter as long as the execution is in mutual exclusion. The following example shows how two EDTs may share access to a data block in `RW` mode, while one

EDT requires `EW` access. In this situation the programmer cannot assume the order in which the EDTs are executed. It might be that EDT1 and EDT2 are executed simultaneously or independently, while EDT3 happens either before, after or in between the others.

A.6.1. Code example

```
/* Example usage of EW (Exclusive-Write)
 * data block access mode in OCR
 *
 * Implements the following dependence graph:
 *
 *        mainEdt
 *        [ DB ]
 *       / |    \
 * (RW)/  |(RW)  \(EW)
 *    /   |       \
 * EDT1  EDT2    EDT3
 *    \   |      /
 *     \  |     /
 *      \ |    /
 *       [ DB ]
 *     shutdownEdt
 *
 */

#include "ocr.h"

#define NB_ELEM_DB 20

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 * data = (u64 *) depv[3].ptr;
    u32 i = 0;
    while (i < NB_ELEM_DB) {
        ocrPrintf("%"PRIu64" ",data[i]);
        i++;
    }
    ocrPrintf("\n");
    ocrDbDestroy(depv[3].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t writerEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 * data = (u64 *) depv[0].ptr;
    u64 lb = paramv[0];
    u64 ub = paramv[1];
    u64 value = paramv[2];
    u32 i = lb;
    while (i < ub) {
        data[i] += value;
        i++;
    }
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    void * dbPtr;
    ocrGuid_t dbGuid;
    u32 nbElem = NB_ELEM_DB;
    ocrDbCreate(&dbGuid, &dbPtr, sizeof(u64) * NB_ELEM_DB, 0, NULL_HINT, NO_ALLOC);
    u64 i = 0;
```

```
    int * data = (int *) dbPtr;
    while (i < nbElem) {
58      data[i] = 0;
        i++;
    }
    ocrDbRelease(dbGuid);

63  ocrGuid_t shutdownEdtTemplateGuid;
    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0, 4);
    ocrGuid_t shutdownGuid;
    ocrEdtCreate(&shutdownGuid, shutdownEdtTemplateGuid, 0, NULL, EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_HINT, NULL);
68  ocrAddDependence(dbGuid, shutdownGuid, 3, DB_MODE_CONST);

    ocrGuid_t writeEdtTemplateGuid;
    ocrEdtTemplateCreate(&writeEdtTemplateGuid, writerEdt, 3, 2);

73  ocrGuid_t eventStartGuid;
    ocrEventCreate(&eventStartGuid, OCR_EVENT_ONCE_T, false);

    // RW '1' from 0 to N/2 (potentially concurrent with writer 1, but different range)
    ocrGuid_t oeWriter0Guid;
78  ocrGuid_t writer0Guid;
    u64 writerParamv0[3] = {0, NB_ELEM_DB/2, 1};
    ocrEdtCreate(&writer0Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv0,
         EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_HINT, &oeWriter0Guid);
    ocrAddDependence(oeWriter0Guid, shutdownGuid, 0, false);
83  ocrAddDependence(dbGuid, writer0Guid, 0, DB_MODE_RW);
    ocrAddDependence(eventStartGuid, writer0Guid, 1, DB_MODE_CONST);

    // RW '2' from N/2 to N (potentially concurrent with writer 0, but different range)
    ocrGuid_t oeWriter1Guid;
88  ocrGuid_t writer1Guid;
    u64 writerParamv1[3] = {NB_ELEM_DB/2, NB_ELEM_DB, 2};
    ocrEdtCreate(&writer1Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv1,
         EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_HINT, &oeWriter1Guid);
    ocrAddDependence(oeWriter1Guid, shutdownGuid, 1, false);
93  ocrAddDependence(dbGuid, writer1Guid, 0, DB_MODE_RW);
    ocrAddDependence(eventStartGuid, writer1Guid, 1, DB_MODE_CONST);

    // EW '3' from N/4 to 3N/4
    ocrGuid_t oeWriter2Guid;
98  ocrGuid_t writer2Guid;
    u64 writerParamv2[3] = {NB_ELEM_DB/4, (NB_ELEM_DB/4)*3, 3};
    ocrEdtCreate(&writer2Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv2,
         EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_HINT, &oeWriter2Guid);
    ocrAddDependence(oeWriter2Guid, shutdownGuid, 2, false);
103 ocrAddDependence(dbGuid, writer2Guid, 0, DB_MODE_EW);
    ocrAddDependence(eventStartGuid, writer2Guid, 1, DB_MODE_CONST);

    ocrEventSatisfy(eventStartGuid, NULL_GUID);

108 return NULL_GUID;
}
```

A.7. Acquiring contents of a data block as a dependence input

This example illustrates the usage model for accessing the contents of a data block. The data contents of a data block are made available to the EDT through the input slots in depv. The input slots contain two fields: the GUID of the data block and pointer to the contents of the data block. The runtime process grabs a pointer to the contents through a process called “acquire”. The acquires of all data blocks accessed inside the EDT have to happen before the EDT starts execution. This implies that the runtime requires knowledge of which data blocks it needs to acquire. That information is given to the runtime through the process of dependence satisfaction. As a result, a data block’s contents are available to the EDT only if that data block has been passed in as the input on a dependence slot or if the data block is created inside the EDT.

A.7.1. Code example

```
/* Example to show how DB guids can be passed through another DB.
 * Note: DB contents can be accessed by an EDT only when they arrive
 * in a dependence slot.
 *
 * Implements the following dependence graph:
 *
 *      mainEdt
 *      [ DB1 ]
 *         |
 *        EDT1
 *         |
 *      [ DB0 ]
 *    shutdownEdt
 *
 */

#include "ocr.h"

#define VAL 42

ocrGuid_t exampleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t *dbPtr = (ocrGuid_t*)depv[0].ptr;
    ocrGuid_t passedDb = dbPtr[0];
    ocrPrintf("Passing DB: "GUIDF"\n", GUIDA(passedDb));
    ocrDbDestroy(depv[0].guid);
    return passedDb;
}

ocrGuid_t awaitingEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 i;
    u32 *dbPtr = (u32*)depv[0].ptr;
    ocrPrintf("Received: %"PRIu32"\n", dbPtr[0]);
    ocrDbDestroy(depv[0].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u32 i;
```

```
41      // Create DBs
        u32* ptr0;
        ocrGuid_t* ptr1;
        ocrGuid_t db0Guid, db1Guid;
        ocrDbCreate(&db0Guid, (void**)&ptr0, sizeof(u32), DB_PROP_NONE, NULL_HINT, NO_ALLOC);
46      ocrDbCreate(&db1Guid, (void**)&ptr1, sizeof(ocrGuid_t), DB_PROP_NONE, NULL_HINT,
            NO_ALLOC);
        ptr0[0] = VAL;
        ptr1[0] = db0Guid;
        ocrPrintf("Sending: %"PRIu32" in DB: "GUIDF"\n", ptr0[0], GUIDA(db0Guid));
        ocrDbRelease(db0Guid);
51      ocrDbRelease(db1Guid);

        // Create Middle EDT
        ocrGuid_t exampleTemplGuid, exampleEdtGuid, exampleEventGuid;
        ocrEdtTemplateCreate(&exampleTemplGuid, exampleEdt, 0 /*paramc*/, 1 /*depc*/);
56      ocrEdtCreate(&exampleEdtGuid, exampleTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
            EDT_PROP_NONE, NULL_HINT, &exampleEventGuid);

        // Create AWAIT EDT
        ocrGuid_t awaitingTemplGuid, awaitingEdtGuid;
61      ocrEdtTemplateCreate(&awaitingTemplGuid, awaitingEdt, 0 /*paramc*/, 1 /*depc*/);
        ocrEdtCreate(&awaitingEdtGuid, awaitingTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
            NULL,
            EDT_PROP_NONE, NULL_HINT, NULL);
        ocrAddDependence(exampleEventGuid, awaitingEdtGuid, 0, DB_DEFAULT_MODE);

66      // START Middle EDT
        ocrAddDependence(db1Guid, exampleEdtGuid, 0, DB_DEFAULT_MODE);

        return NULL_GUID;
    }
```

A.7.1.1. Details

The mainEdt creates two data blocks (`db0Guid` and `db1Guid`). It then sets the content of `db0Guid` to be an user-defined value, while the content of `db1Guid` is set to be the GUID value of `db0Guid`. The runtime then creates an EDT (`exampleEdtGuid`) that takes one input dependence. It creates another EDT (`awaitingEdtGuid`) and makes it dependent on the satisfaction of the `exampleEdtGuid`'s output event (`exampleEventGuid`). Finally, mainEdt satisfies the dependence of `exampleEdtGuid` with the data block `db1Guid`.

Once `exampleEdtGuid` starts executing function "exampleEdt", the contents of `db1Guid` are read. The function then retrieves the GUID of the data block `db0Guid` from the contents of `db1Guid`. Now in order to read the contents of `db0Guid`, the function satisfies the output event with `db0Guid`.

Inside the final EDT function "awaitingEdt", the contents of `db0Guid` can be read. The function prints the content read from the data block and finally calls "ocrShutdown".

B. OCR API Extensions

This chapter describes API extensions to OCR that are being considered for inclusion in the specification. The extensions described in this chapter may not be fully functional on all platforms. Feedback on the usefulness of these extensions as well as any modifications to them is welcomed and should be emailed to the OCR User mailing list (ocr-user@eci.exascale-tech.com).

B.1. User specified hints

OCR allows application developers to provide hints about the application using an extension of the standard API. The motivation of the hints API is to enable the transfer of knowledge, which is usually lost when expressing an program using the standard OCR API. Hints never impact the correctness of a program but are instead "extra" information that the runtime can use, for example, to improve the allocation of data blocks and scheduling of EDTs. Note that an OCR implementation can ignore all hints or selectively choose the ones to respect.

B.1.1. OCR hint framework

- **`ocrHint_t`**: Hints in OCR use a special type called **`ocrHint_t`**. A variable of this type can be declared inside an EDT.
- **`ocrHintType_t`**: The hint variable has to be initialized for a specific usage *type*. Currently, there are four types of hints:
  - **`OCR_HINT_EDT_T`** is used to initialize the hint variable to be used for EDT specific hints. EDT specific hints can also be used for EDT templates. When an EDT is created, it is initialized with the hints that are set on the EDT template.
  - **`OCR_HINT_DB_T`** is used for data block specific hints.
  - **`OCR_HINT_EVT_T`** is used for event specific hints.
  - **`OCR_HINT_GROUP_T`** is used for group specific hints. OCR groups are logical entities to which multiple OCR objects can be associated. Providing a hint for a group will help the runtime guide the scheduling of these OCR objects. Group hints can be applied either to each individual object in the group or to the group as a whole.
- **`ocrHintVal_t`**: The type of a hint object's value can vary depending on the hint being set; therefore, values are provided via **`ocrHintVal_t`**, which is defined as a union with at least the following fields:
  - `s64 s64Value`
  - `ocrGuid_t guidValue`

  In the future, this union may be expanded to provide more value-type options.
- **`ocrHintProp_t`**: User hints are set as hint properties in the hint variable. The runtime defines the hint properties that can be set for a specific hint type. Supported properties are an ongoing developement. Please refer to ocr-types.h for all the currently supported hint properties.

After the variable has been initialized, it can start accepting hint properties. The hint variable can then be used to transfer all the set properties to an OCR object guid.

It is also possible for the user to read the hints that are set on a specific OCR object. The user may then choose to update the values of the properties on the guid.

Functions

- u8 **ocrHintInit**(ocrHint_t *hint, ocrHintType_t hintType)

  Initializes a user hint variable
- u8 **ocrHintSetValue**(ocrHint_t *hint, ocrHintProp_t hintProp, ocrHintVal_t value)

  Sets the value of a specific hint property
- u8 **ocrHintUnsetValue**(ocrHint_t *hint, ocrHintProp_t hintProp)

  Unsets the value of a specific hint property
- u8 **ocrHintGetValue**(ocrHint_t *hint, ocrHintProp_t hintProp, ocrHintVal_t *value)

  Gets the value of a specific hint property
- u8 **ocrSetHint**(ocrGuid_t guid, ocrHint_t *hint)

  Sets all hint properties defined in the hint variable to the target guid
- u8 **ocrGetHint**(ocrGuid_t guid, ocrHint_t *hint)

  Gets the existing hint properties from a specific guid

B.1.2. u8 ocrHintInit(ocrHint_t * *hint*, ocrHintType_t *hintType*)

Initializes a user hint variable of a specific type. The hint variable has to be initialized before any hint properties can be set.

Parameters

in	**hint**	The hint variable to be initialized.
in	**hintType**	The usage type for the hint variable

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): error in initializing hint when hintType is unrecognized

Description User hint objects in OCR are of type **ocrHint_t**. A user hint object is a stack allocated variable that can only exist within the lifetime of an EDT. The user has to first call *ocrHintInit* to initialize a local hint variable before using it. Subsequent calls to ocrHintInit will reset the variable to an empty state, that is where none of the properties are set.

B.1.3. u8 ocrHintSetValue(ocrHint_t * *hint*, ocrHintProp_t *hintProp*, ocrHintVal_t *value*)

Sets the value of a specific hint property on to a hint variable. Multiple hint properties of the same usage type can be set on a single hint variable.

Parameters

in	**hint**	The hint variable for which hints are set.
in	**hintProp**	The hint property to set
in	**value**	The value of the hint property

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If property is invalid for the hint type.

Description A value for a specific hint property can be set if that property is supported for that specific hint type. If a property value is already set, then the value is updated.

B.1.4. u8 ocrHintUnsetValue(ocrHint_t * *hint*, ocrHintProp_t *hintProp*)

Unsets the value of a specific hint property in a hint variable.

Parameters

in	**hint**	The hint variable for which hints are set.
in	**hintProp**	The hint property to unset

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If property is invalid for the hint type.

Description A value for a specific hint property can be unset if that property is supported for that specific hint type and is already set. If a property value was not set earlier, then unset is a no-op.

B.1.5. u8 ocrHintGetValue(ocrHint_t * *hint*, ocrHintProp_t *hintProp*, ocrHintVal_t * *value*)

Gets the value of a specific hint property from a hint variable.

Parameters

in	**hint**	The hint variable from which to get hints
in	**hintProp**	The hint property to get
out	**value**	The value of the hint property read from the hint variable

Gets the value of a specific hint property that was already set on the hint variable.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If property is invalid for the hint type.
- OCR_ENOENT (**immediate**): If property is not set on this hint variable

Description A value for the hint property can be gotten from a hint object if the property is supported and it has already been set.

B.1.6. u8 ocrSetHint(ocrGuid_t *guid,* ocrHint_t * *hint*)

Sets all the hint properties in the hint variable on to the target OCR object guid. The target guid has to be of a compatible type with the hint variable's usage type.

Parameters

in	**guid**	The target guid of the hints
in	**hint**	The hint variable used to set the hint properties

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**deferred**): If hint type and target guid kind are incompatible
- OCR_EFAULT (**deferred**): Some hints were not set due to insufficient or invalid values of hint properties

Description If the target guid has some of the hints already set then their values are updated. If hints are set on the same guid concurrently, then the final values are undefined.

B.1.7. u8 ocrGetHint(ocrGuid_t *guid,* ocrHint_t * *hint*)

Gets the existing hint properties that have been set on a specific guid.

Parameters

in	**guid**	The target guid of the hints
out	**hint**	The hint variable that will be populated with the target guid's existing hints

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If hint type and target guid kind are incompatible

Description The hint variable that is used as the output argument will be updated with hint property values from the guid. The type of the hint variable should be compatible with the guid kind. If the hint properties that exist on the guid are already set on the hint variable then those properties will be updated. The hint properties that are present on the guid but not on the hint variable will be added to the hint variable. If the hint variable has other properties set which do not exist on the guid, those properties will be retained in the hint variable.

B.1.8. Usage scenarios

As an example, we show how hints can be added to an OCR program solving a tiled cholesky factorization problem.

```
//Here we set hints on the EDT template for the various tasks:
//In this example, we show how to set the default affinity of all EDTs
//being generated out of these EDT templates will be set to the DB that
//is passed in to slot 0. That is why we set the value of the hint property
//OCR_HINT_EDT_SLOT_MAX_ACCESS to be 0

ocrHint_t hintVar;
ocrHintInit(&hintVar, OCR_HINT_EDT_T);
ocrHintVal_t slotNumberHintValue = { .s64Value = 0 };
if (ocrHintSetValue(&hintVar, OCR_HINT_EDT_SLOT_MAX_ACCESS, slotNumberHintValue) == 0) {
    ocrSetHint(templateSeq, &hintVar);
    ocrSetHint(templateTrisolve, &hintVar);
    ocrSetHint(templateUpdateNonDiag, &hintVar);
    ocrSetHint(templateUpdate, &hintVar);
}
```

B.2. Labeled GUIDs

GUIDs are used to identify OCR objects and are opaque to the programmer. A consequence of this opacity is that if two EDTs need to use a common object, they both need to have a-priori knowledge of the GUID for that object. If the object was created much earlier in the execution flow, both EDTs therefore need to have the GUID passed down either through data blocks or parameters. This is inconvenient and can lead to a glut of parameters and data blocks solely dedicated to passing down GUIDs.

Labeled GUIDs provide a mechanism by which a programmer can reason about GUIDs; an API is provided to "translate" a programmer defined index range into GUIDs. The transformation is such that all EDTs invoking this API with the same input will get the same resulting GUID. In effect, EDTs no longer need to agree on an opaque GUID (which requires a-priori knowledge) but only on a common index which can be achieved only through semantic knowledge of the application. Concretely, this is the difference between "the 'neighbor' EDT you need to communicate with has GUID X" and "give me X, the GUID of my neighbor Y".

B.2.1. Usage scenarios

Several usage scenarios have been identified for labeled GUIDs. These scenarios are by no means exhaustive but have driven the current design.

B.2.1.1. Referring to a previously created OCR object

In this scenario, a root EDT `R` creates a sink EDT `S` (for example a reduction EDT) and then spawns multiple child EDTs which will in turn spawn EDTs which will satisfy a slot of `S`. Without labeled GUIDs, `S`'s GUID would need to be passed down to each and every producer. Labeled GUIDs allow the producers to ask the runtime for `S`'s GUID.

B.2.1.2. Unsynchronized object creation

Traditionally, if an EDT wants to refer to an OCR object, that object's creation needs to have happened before its use, and, conversely, the object's eventual destruction needs to happen after its use. In a situation where two EDTs `A` and `B` need to use an object, that object's creation needs to happen in a third EDT `C` which happened before `A` and `B`. In other words, there is a dependence chain between `C` and `A` as well as one between `C` and `B`.

This behavior is not always desired. For example, suppose an algorithm where, at each iteration, each EDT creates its "clone" for the next iteration; in other words, the algorithm avoids a global barrier between iterations. Suppose that within an iteration, an EDT `B` depends on another EDT `A`. Without labeled GUIDs, `B` and `A` would have no way on agreeing on the event to use to

synchronize. Labeled GUIDs allow both A and B to "create" the event and the runtime will guarrantee that **a)** only one event is created and **b)** both A and B get the same GUID for that event.

B.2.2. API

The following enum is used to specify the types of objects a GUID can refer to: **enum ocrGuidUserKind** containing:

- **GUID_USER_NONE** The GUID is invalid and does not refer to any object.
- **GUID_USER_DB** The GUID refers to a data block.
- **GUID_USER_EDT** The GUID refers to an EDT.
- **GUID_USER_EDT_TEMPLATE** The GUID refers to an EDT template.
- **GUID_USER_EVENT_ONCE** The GUID refers to a ONCE event.
- **GUID_USER_EVENT_IDEM** The GUID refers to an IDEMPOTENT event.
- **GUID_USER_EVENT_STICKY** The GUID refers to a STICKY event.
- **GUID_USER_EVENT_LATCH** The GUID refers to a LATCH event.

The primary functions supporting labeled GUIDs are listed below. These functions allow the user to reserve a flat range of GUIDs, and then retrieve a particular GUID by its index in that range. The user may decide to map an N-dimensional tuple space onto the flat index space for a labeled GUID range.

- u8 **ocrGuidRangeCreate**(ocrGuid_t *rangeGuid, u64 guidCount, ocrGuidUserKind kind)

  Reserves a range of GUIDs to be used by the labeling mechanism.
- u8 **ocrGuidRangeDestroy**(ocrGuid_t rangeGuid)

  Destroy a GUID range as created by ocrGuidRangeCreate.
- u8 **ocrGuidFromIndex**(ocrGuid_t *outGuid, ocrGuid_t rangeGuid, u64 idx)

  Converts an index into a GUID. This function is used with GUID ranges created using ocrGuidRangeCreate.
- u8 **ocrGetGuidKind**(ocrGuidUserKind *outKind, ocrGuid_t guid)

  Gets the type from a GUID. This can be used to indicate if the GUID refers to a valid object.

B.2.2.1. u8 ocrGuidRangeCreate(ocrGuid_t * *rangeGuid,* u64 *guidCount,* ocrGuidUserKind *kind*)

Creates a new instance of a GUID range which can be used to map indices (from 0 to **guidCount**) to GUIDs.

Parameters

out	**rangeGuid**	On successful creation, contains the GUID of the range created. This GUID should be used with **ocrGuidFromIndex**. If the call fails, the returned value is undefined.
in	**guidCount**	The number of GUIDs to reserve in this range.
in	**kind**	Kind of the GUIDs stored in this range.

Returns 0 if no immediate error was detected. There are no other defined error codes for this call.

Description The **rangeGuid** returned by this function should be used with ocrGuidFromIndex.

B.2.2.2. u8 ocrGuidRangeDestroy(ocrGuid_t * *rangeGuid*)

Destroys a range created by ocrGuidRangeCreate.

Parameters

in	**rangeGuid**	GUID of the range to destroy.

Returns 0 if no immediate error was detected. There are no other defined error codes for this call.

Description This function does not affect any of the GUIDs that have already been created with the range or in the range. It does, however, un-reserve all the ones that have been reserved but not used.

B.2.2.3. u8 ocrGuidFromIndex(ocrGuid_t * *outGuid,* ocrGuid_t *rangeGuid,* u64 *idx*)

Uses the range created using ocrGuidRangeCreate and referenced by **rangeGuid** to convert **idx** into a GUID.

Parameters

out	**outGuid**	GUID corresponding to the index.
in	**rangeGuid**	GUID of the range to use.
in	**idx**	Index to convert to a GUID.

Returns 0 if no immediate error was detected. There are no other defined error codes for this call.

Description This function assumes that the programmer has already calculated the target index in the GUID range. This index is then used to index into the GUID space reserved by the ocrGuidRangeCreate call.

B.2.2.4. u8 ocrGetGuidKind(ocrGuidUserKind * *outKind,* ocrGuid_t *guid*)

This function returns the type of OCR object (event, data block, EDT) that the GUID refers to or **OCR_GUID_NONE** if the GUID is invalid.

Parameters

out	**outKind**	The kind of object this GUID refers to.
in	**guid**	The GUID to get information from.

Returns 0 if no immediate error was detected. There are no other defined error codes for this call. Note that returning **OCR_GUID_NONE** is considered a successful execution.

Description With labeled GUIDs, having a GUID does not necessarily mean that it refers to a valid object. This function addresses this concern by determining if a GUID refers to a valid object. Note that the information returned may be stale if concurrent creation/destruction of the object is happening.

B.2.3. Other API changes

The creation calls are all modified to allow them to accept a GUID as input (as opposed to just output). In the current implementation, events, EDTs and data blocks can be labeled. There are some restrictions on using labeled EDTs, namely: **a)** labeled EDTs cannot have dependences listed in the create call, **b)** labeled EDTs cannot request an output event and **c)** if an instance of a labeled EDT is created within a finish scope, that instance will only be registered in one of the finish scopes if the instance is created in multiple places. To use labeled GUIDs, the programmer should pass in the GUID returned by ocrGuidFromIndex as the first argument of an **ocrXXCreate** call and also add an additional flag to the properties field of that call:

- **GUID_PROP_IS_LABELED** The input GUID to the call should be used as the GUID for the created object. Note that with this flag, the user is responsible for ensuring that only one EDT creates the object (in other words, this is a "trust me" mode) where the runtiem incurs very little cost to creating the object.
- **GUID_PROP_CHECK** Similar to **GUID_PROP_IS_LABELED**, this property will also cause the use of labeled GUIDs. However, the runtime ensures that the object is only created once. In other words, other EDTs trying to create the same object (same GUID) will get an error code and know that the object has already been created.
- **GUID_PROP_BLOCK** This property blocks the creation call until the object no longer exists and can therefore be recreated. This property is not in line with the non-blocking philosophy of OCR but is there to support legacy programming models.

B.2.4. Other considerations

This extension is a work in progress. For example, one issue with this proposal (and with many of the other creation calls) is that the creation of the GUID may be delayed and require communication. This is particularly true with labeled GUIDs as the runtime is constrained in the GUID is can use to create an object. One proposal is to have the notion of a local identifier which could only be used inside an EDT. This would allow creation calls to return immediately and allow the runtime to defer all long-latency calls till after the EDT finishes.

Normally, the exact definition of the point in time that an object is destroyed is not necessary. The object cannot be used after the destroy call was made or after it was supposed to be destroyed automatically. It is up to the runtime to determine the right time to reclaim the object's resources. However, if the object is created using a labeled GUID, it is possible to re-use the GUID to create another object. When the create call is made with **GUID_PROP_CHECK**, the outcome depends on the state of the object. A new object is only created if an object with the same GUID was not created elsewhere or if any such object was already destroyed. It's therefore necessary to establish a clear relation between the destruction of the object and the create call. For the purpose of labeled GUIDs, all objects are considered to be completely destroyed by a call to the appropriate destroy function or at the moment they should be automatically destroyed. EDTs are exception to this rule. An EDT releases its labeled GUID for re-use at some point after it becomes ready, but before it starts running.

B.3. Parameterized event creation

This extension allows the user to provide an additional parameter at the creation of an event to customize its behavior. For example, with latch events, the user may specify an initial count for the latch.

B.3.1. Usage scenarios

This extension makes the initialization of latch events simpler. Previously the programmer had to write a loop to increment or decrement the counter to reach a certain value. The same is now achieved by passing the counter value as part of the extra parameter.

B.3.2. API

A new API call named **`ocrEventCreateParams`** is added. It has the same signature as **`ocrEventCreate`** with an additional argument of type **`ocrEventParams_t*`**. The later is a struct composed of a union of sub-structure declarations that are referenced by name, one for each type of event that can be configured. Note a runtime implementation can make no assumptions about the lifetime of the passed pointer and must ensure that the parameter can be safely destroyed by the user after returning from the call.

The following parameters are available for latch events:

- **`EVENT_LATCH`**:
  - **`u64 counter`**: Initial value of the latch counter.

Functions

- u8 **`ocrEventCreateParams`**(ocrGuid_t * guid, ocrEventTypes_t eventType, u16 properties, const ocrHint_t *hint, const ocrEventParams_t *params)

  DOC TODO

B.3.3. u8 ocrEventCreateParams(ocrGuid_t * *guid,* ocrEventTypes_t *eventType,* u16 *flags,* const ocrHint_t * *hint,* const ocrEventParams_t * *params*)

Requests the creation of an event of the specified type, initialized with the provided parameters. After a successful call, the runtime will return the GUID for the newly created event. The returned GUID is immediately usable.

Parameters

out	**guid**	On successful creation, contains the GUID of the event. If the call fails, the returned value is undefined.
in	**eventType**	The type of event to create. See enum ocrEventTypes_t.
in	**flags**	Flags impacting the creation of the event. Currently, the following flags are supported: • EVT_PROP_NONE: Default behavior • EVT_PROP_TAKES_ARG: The created event will potentially carry a data block on satisfaction.
in	**hint**	Reserved for future use. This parameter should be NULL_HINT.
in	**params**	Parameters to initialized the event with.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_ENOMEM (**deferred**): The runtime could not create the event due to insufficient memory
- OCR_EINVAL (**immediate**): The **eventType** argument is invalid or incompatible with **flags**

Description This function is used to create the basic synchronization mechanism is OCR: the event. The lifetime of the created event is dependent on its type. See Section 1.4.3 for more details.

B.4. Counted events

A *counted event* will destroy itself automatically after both of the following conditions are true: **a)** it has been satisfied and **b)** a pre-determined number of OCR objects have a dependence on the event. In other words, a counted event is like a once event in the sense that it auto-destroys once satisfied but is safer than a once event because it will expect to have a certain number of dependences waiting on it and will ensure that those dependences are satisfied before destroying itself. This eliminates the necessity for programmers to ensure that all add dependences that have the once event as a source happen before satisfying the event.

Counted events trigger immediately when satisfied. Precisely, a satisfy call triggers the event immediately which will, in turn, satisfy any dependence already registered at that time. Calls to **ocrAddDependence** that happen after the satisfy call, immediately turn into satisfy calls themselves. The event is destroyed when it has been satisfied and the count of expected dependences is reached.

B.4.1. API

Counted events extend the **`enum ocrEventTypes_t`** type with a OCR_EVENT_COUNTED_T declaration. Counted events rely on the parameterized event creation extension (see Section B.3 to allow the programmer to specify the number of dependences the counted event will have. A counted event's parameters are accessible under the EVENT_COUNTED field of the ocrEventParams_t type.

The following parameters are available for counted events:

- **`EVENT_COUNTED`**:
  - **`u64 nbDeps`**: Expected number of dependences to be added to the event.

Note it is an error to create a counted event with zero dependences.

B.5. Channel events

Channel events represent a whole new class of OCR events that can trigger multiple times. We define a *generation* as the time between consecutive triggers of the event (the first generation is defined as the time between the event creation and its first trigger).

Channel events have a different triggering rule; a channel event will trigger when both the following conditions are met: **a)** the event has been satisfied a certain number of times and **b)** a certain number of dependences has been added to the event. Once a channel event triggers, the event is reset and a new generation starts.

Channel events also preserve FIFO ordering when there is a sequenced-before or happen-before relationship between two satisfy calls made on the same channel event GUID. Note that the ordering is only guaranteed when using the GUID of the channel event; any indirection through other types of events may break the ordering.

B.5.1. Usage scenarios

The current implementation is restricted and defines a generation as one satisfy paired with one dependence being added. This extension is meant to explore the usefulness of channel events before a more general model is defined and implemented. The implementation provides a 'window' of generations in the form of an internal bounded buffer for which the size is known at the creation of the channel event. This essentially translates into the implementation being able to buffer a number of satisfy calls that do not yet have a matching dependence registered (and vice-versa) up to the bound value. In order to not exceed the bound, the programmer must enforce proper synchronization.

The current implementation is geared toward use-cases found in domain-decomposition applications where each domain communicates its halo to its neighbor at each iteration of the algorithm. A typical implementation is to represent each sub-domain and iteration as an EDT that depends on a set of data blocks for its own domain as well as data blocks corresponding to contributions from its neighbor domains. In such situations, where the communication pattern is static, it is beneficial to setup a topology of channel events once during the startup of the application and keep reusing the same event GUIDs to satisfy the neighbors' upcoming EDT iteration instances.

B.5.2. API

Channel events extend the **`enum ocrEventTypes_t`** type with an OCR_EVENT_CHANNEL_T declaration. Channel events also rely on the parameterized event creation extension to allow the programmer to customize the type of channel event. A channel event's parameters are accessible under the EVENT_CHANNEL field of the ocrEventParams_t type. Note the actual parameters and their semantic are still under active development.

The following parameters are available for channel events:

- **`EVENT_CHANNEL`**:
  - **u32 `maxGen`**: The maximum number of generations the event can buffer.
  - **u32 `nbSat`**: The number of satisfy required to trigger. Currently must be set to 1.
  - **u32 `nbDeps`**: The number of dependences required to trigger. Currently must be set to 1.

B.6. EDT Local Storage

This extension provides a dedicated portion of fixed-size pre-allocated memory made available to each EDT for its private use, analogous to TLS (Thread Local Storage). They are typically used by user-level libraries and high-level language translators to maintain per-EDT state in a uniform manner.

The extension proposes the use of the following API:

B.6.1. u8 ocrEdtLocalStorageGet(void ** *ptr,* u64 * *elsSize*)

Exposes the local data-store associated with the current EDT and its size.

Parameters

out	**ptr**	Pointer to the local data-store.
out	**elsSize**	Size of the local data-store

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If the pointer is invalid.

B.7. EDT Query

The extension provides interfaces to allow the application code in EDTs to query themselves. These are currently used for application's diagnostics and debugging.

This extension provides two API calls as detailed below.

B.7.1. u8 ocrCurrentEdtGet(void * *curEdt*)

Exposes the GUID of the executing EDT from which this function is called.

Parameters

out	**curEdt**	GUID of the executing EDT.

Returns 0 if no immediate error was detected or the following error codes:

- 0: Successful
- OCR_EINVAL (**immediate**): If the pointer is invalid.

B.7.2. u8 ocrCurrentEdtOutputGet(void * *outputEvent*)

Exposes the GUID of the output event of the executing EDT from which this function is called.

Parameters

out	**outputEvent**	GUID of the executing EDT's output event.

Returns 0 if no immediate error was detected or the following error codes:

- OCR_EINVAL (**immediate**): If the pointer is invalid.

C. Implementation Notes

This appendix contains details on the ways in which the various OCR reference implementations differ from the specification. This section will keep evolving as the implementations become more compliant.

C.1. General notes

This section describes the limitations common to all OCR implementations.

Data block access modes The data block access modes are only supported using the `l`ockableDB implementation of data blocks. The `r`egularDB implementation ignores the modes. Furthermore, the read-only mode has not been fully tested.

D. OCR Change History

September 2014 Release of OCR 0.9 including the first version of this specification.

April 2015 Release of OCR 0.95. Fixed some typos in the spec and cleaned up some subtle flaws in the memory model.

June 2015 Release of OCR 1.0.0. Restructured the specification for clarity and updated the API to make the names of the memory modes more intuitive. Also moved the API documentation from doxygen to human-readable TEX with proper specification language.

September 2015 Release of OCR 1.0.1. Added Section B on labeled GUIDs and hints. Minor other clarifications.

March 2016 Release of OCR 1.1.0 and 1.0.2. Notable changes include:

- API changes for `ocrDbCreate` and `ocrEdtCreate` to use hints instead of affinities
- Added GUID management section (2.5)
- Added counted and channel event extensions
- Clarification on API return codes, multiple acquire of the same data block
- Added section on version numbers

December 2016 Release of OCR 1.2.0 (Candidate). This release breaks certain APIs thus the naming change to 1.2. This is a candidate release as no implementations fully implement 1.2.0 as of December 2016. Notable changes include:

- Added the "downgrade release" of data blocks
- Added the possibility to use hints when creating events
- Added the possibility to self-identify an EDT (and its output event)
- Added the possibility of specifying an output event when creating an EDT
- Made the API calls more "asynchronous" friendly in particular by: **a)** removing the option of using EDT_PARAM_DEF, and **b)** relaxing the requirements on the values returned through error codes.
- Added several clarifications, in particular on: **a)** OCR object lifetimes, **b)** zero-sized data blocks, **c)** behavior of EDTs destroyed in a finish scope, and **d)** the flags during event creation.

- Clarified the behavior of several extensions, namely: **a)** hints, **b)** labeled GUIDs, and **c)** the ELS.