



OCR

The Open Community Runtime Interface

Version 0.9, September, 2014

Editors: Tim Mattson, Rob van der Wijngaart, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Romain Cledat, Bala Seshasayee, Vivek Sarkar

Copyright © 2014 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

1	Introduction	1
1.1	Scope	2
1.2	Glossary	2
1.3	Execution Model	5
1.3.1	OCR Platform	6
1.3.2	OCR objects	6
1.3.3	Trigger rule	9
1.3.4	OCR program execution	11
1.4	Memory Model	13
1.5	Organization of this document	15
2	OCR API Documentation	16
2.1	Module Index	16
2.1.1	Modules	16
2.2	Data Structure Index	16
2.2.1	Data Structures	16
2.3	Module Documentation	17
2.3.1	OCR extensions/experimental APIs	17
2.3.2	Affinity extension	17
2.3.3	OCR used as a library extension	19
2.3.4	Interface for runtimes built on top of OCR	21
2.3.5	Data-block management for OCR	23
2.3.6	Event Management for OCR	29
2.3.7	Event Driven Task API	32

2.3.8	OCR dependence APIs	35
2.3.9	Limited "standard" API for OCR	36
2.3.10	Types and constants used in OCR	38
2.3.11	General types and constants	42
2.3.12	Types and constants associated with data blocks	43
2.3.13	Types and constants associated with EDTs	45
2.3.14	Types and constants associated with events	47
2.3.15	Support calls for OCR	49
2.4	Data Structure Documentation	50
2.4.1	ocrConfig_t Struct Reference	51
2.4.2	ocrEdtDep_t Struct Reference	52
A	OCR Examples	55
A.1	OCR's "Hello World!"	55
A.1.1	Code example	55
A.2	Expressing a Fork-Join pattern	56
A.2.1	Code example	56
A.3	Expressing unstructured parallelism	58
A.3.1	Code example	58
A.4	Using a Finish EDT	61
A.4.1	Code example	61
A.5	Accessing a DataBlock with "Intent-To-Write" Mode	65
A.5.1	Code example	65
A.6	Accessing a DataBlock with "Exclusive-Write" Mode	67
A.6.1	Code example	67
A.7	Acquiring contents of a DataBlock as a dependence input	70
A.7.1	Code example	71
B	OCR Change History	73

1. Introduction

Extreme scale computers (such as proposed Exascale computers) contain so many components that the aggregate mean-time-to-failure is small compared to the runtime of an application.

Programming models (and the supporting compilers and runtime systems) must therefore support a variety of features unique to these machines:

- The ability for a programmer to express $O(\text{billion})$ 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 research projects in progress to develop a runtime system for extreme scale computers. This specification describes one of these research runtime systems: the Open Community Runtime or OCR. The central idea behind OCR is to decouple the work of a computation from the “Units of execution” that carry out computations and to decouple the data used in a computation from a computer’s memory. The result is to virtualize the work and data associated with a computation so they can be relocated within a scalable computer to optimize execution of the parallel program. More importantly, virtualization allows a program to respond to hardware issues by relocating work and data from failed components onto working components [17].

The fundamental idea behind OCR is to consider a computation as a dynamically created graph of tasks. Task execution is managed by events; when the events a task is dependent upon are satisfied, the task becomes available to be scheduled for execution.

OCR provides a global name space that holds references to tasks, events and data blocks¹, which supports the transparent scheduling and relocation of these entities across hardware resources.

These events can be used to support a wide range of execution models, including data-flow (when events are associated with data-blocks), fork-join (when events enable the execution of post-join

¹Data blocks, described in Section 1.3.3.1 are exposed through a global ID which provides a reference to the data block itself. A task can access the contents of data blocks either **a**) created inside the task and **b**) data blocks whose IDs are listed as requisite input to the task before it is scheduled for execution.

continuations), bulk-synchronous processing (when event trees can be used to build scalable barriers and collective operations), and combinations thereof.

1.1. Scope

OCR is a vehicle to support research on programming models and runtime systems for extreme scale computers [8, 9]. This specification defines the state of OCR at a fixed point in its development. There are several limitations in OCR that will be relaxed as it continues to develop.

OCR is a runtime system and collection of low level Application Programming Interfaces (APIs). 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**².

All parallelism must be specified explicitly in OCR. It does not extract the concurrency in a program on behalf of a programmer. The OCR execution model is a low level model; abstract enough to support relocation of tasks and data to support resiliency or to minimize the energy consumed by a computation, but low level enough to cleanly map onto the hardware of extreme scale computers.

OCR is designed to handle dynamic task driven algorithms expressed in terms of a directed acyclic graph (DAG). In a 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 relatively heavy weight event driven tasks (EDTs). An OCR programmer should 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.

1.2. Glossary

²Reductions can be supported in OCR using an accumulator/reducer approach [3, 13] and **parallel for** can be supported in OCR using a fork-join decomposition similar to the `cilk_for` construct.

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 as well. 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 or data-block 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 <i>pre-slots</i> numbered from 0 to $N - 1$ and one post-slot. Each of the <i>pre-slots</i> associated with an EDT connects to a single OCR object, while the EDT's single <i>post-slot</i> can connect to multiple OCR objects. An EDT becomes runnable when all its pre-slots have been satisfied; the pre-slots determine which data blocks, if any, the EDT may access.
EDT function	The function that defines the code to be executed by an EDT. The function takes as arguments the number of parameters, the actual array of parameters, the number of dependences and the actual array of dependences. <i>Parameters</i> are static 64-bit values known at EDT creation time and <i>dependences</i> are dynamic control or data dependences. The parameter array is copied by value when the EDT is created. The dependences (namely the array of dependences) are determined at runtime and are fully resolved only when the EDT starts to run. The EDT function can optionally return the GUID of a DB or event that will be passed along its "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. For the case of a finish EDT, the EDTs created within its scope (i.e. its child EDTs and further descendants) complete and satisfy their post-slots before the finish EDT can satisfy its post-slot. The result is that any OCR object linked to the post-slot of the finish EDT will by necessity not be scheduled or satisfied until the finish EDT and all EDTs created during its execution have completed.
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 name space visible to all EDTs.

- 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 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.
- OCR object** A reference counted object managed by OCR. *EDTs*, *events*, *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 can no longer be defined by OCR.
- Released** The state of a 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 could 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 post-slot to a destination pre-slot.
- A slot in the *connected* state can be:
- Satisfied: the source of the link is a triggered slot;
 - Triggered: see 'Trigger';
 - Unsatisfied: the source of the link is not a triggered slot.
- 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, 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 trigger the event and therefore cause it to satisfy its post-slot which will in turn 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 “a slot that has been satisfied and has transferred the updated satisfied status along its link(s) is a triggered slot”.

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.

1.3. Execution Model

An OCR program is best understood as a dynamically created, directed acyclic graph (DAG) [14, 15, 18]. The vertices in the graph are OCR objects that define a computation: tasks which perform the actual computation and events which are used to coordinate the activity of other objects. The edges are dependences between objects (i.e. a *link*) which can represent data dependences (events with an associated data block) or pure control dependences (events).

The tasks within the DAG represent the work carried out by an OCR program. Edges impinging on a task define preconditions for the execution of the task [12]. Tasks whose preconditions have been met are *runnable*. Outgoing edges define dependences and triggers for later objects in the graph. OCR tasks are *non-blocking*. This means that once all preconditions on a task have been met, the task becomes runnable, and when it begins to execute; the task will eventually run to completion regardless of the behavior of any other OCR objects.

OCR programs can either run alone or be encompassed in other programs in a library manner. In the following description, “OCR program” refers to either the entire OCR program if it is running alone or the OCR portion of the program if running library mode.

The OCR program logically starts as a single task, dynamically builds the DAG corresponding to the executing program, 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. 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 [2, 4, 6, 10].

We will define the execution model in OCR by starting with a model of the OCR platform. We will then describe the fundamental objects used to define OCR. Finally, we will describe the details of

how an OCR program executes.

1.3.1. OCR Platform

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 has its own private memory³.
- A globally accessible shared namespace 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.

As dependences are met for the tasks in an OCR program's DAG, the tasks become *runnable*. These tasks and any resources required to support their execution are then submitted to *workers* [5] 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 [1], but they do not hold persistent state visible to an OCR program and are logically opaque to OCR constructs.

1.3.2. OCR objects

An OCR object is a reference counted entity managed by OCR. Every OCR object has a globally unique ID (GUID) that is used to identify the object. 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. A destroyed object and any resources associated with the destroyed object are no longer defined. The object is not actually destroyed and the associated resources are not freed until the reference count is zero⁴.

Furthermore, for OCR data blocks, we have two additional states:

1. *Acquired*: the data associated with the data block has become accessible to the acquiring OCR object thereby incrementing the acquired objects reference count.

³By "private" we mean a memory region that is not accessible to other processing elements.

⁴As an optimization, the runtime may choose to reuse of the same physical object for different logical objects [11, 16].

2. *Released*: The object is no longer accessible by the OCR object that had earlier acquired it. The reference count on the released object is decremented.

An OCR program is defined in terms of three fundamental objects.

Event Driven Tasks (EDT) A non-blocking unit of work in an OCR program.

Data blocks (DB) A contiguous block of memory managed by the OCR runtime accessible to any OCR objects to which it is linked.

Events An object to manage dependences between OCR objects and to define ordering relationships (synchronization) between them.

In addition to these fundamental objects, OCR defines a number of associated objects that simplify OCR programming or support specific desired behaviors of the fundamental objects.

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

Affinity container An OCR object used to influence the placement of EDTs in an executing program.

An OCR program defines a graph with the three fundamental OCR objects (EDTs, DBs and Events) as the nodes of the graphs and edges are *links* between objects. A link defines a dependence between OCR objects. The links are defined in terms of *slots* on the OCR object. A slot defines an end point for a dependence for an OCR object.

Event, data blocks and EDTs each have a single *post slot*. used to communicate the state of an OCR object to other OCR objects. For example, if an EDT wanted to let another EDT know that it had finished its assigned work, it could do so by signaling over its post-slot that it is *satisfied* or equivalently, that the post-slot is *triggered*. The rules defining when a post slot triggers, the so-called *post-slot trigger rule* depends on the type of OCR object and is discussed in Section 1.3.3.

Some OCR objects (such as EDTs) can also have an optional set of *pre-slots*. A pre-slot defines an incoming dependence or a pre-condition for execution by an EDT. The post-slot of one EDT, for example, can be connected to the pre-slot of another EDT thereby establishing a control dependence between the EDTs. Likewise, the post-slot of a data block can be connected to the pre-slot of an EDT to establish an immediately satisfied data dependence.

Slots are used along with data block objects to define data dependences between OCR objects. For example, for producer consumer relationships the post slot of the producer EDT can be connected 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 the post-slot and signals its “satisfied” state to the consumer who can then safely begin working with the data block from the producer.

Refer to Section 1.2 for a definition of the states of a Slot. All slots are initially in the **unconnected** state. Data block post slots are immediately **satisfied** as soon as they are connected.

1.3.2.1. EDTs

A task defines the basic unit of work within a programming model. As mentioned earlier, a task is a non-blocking unit of work. Once all pre-conditions on the OCR task have been met, it becomes runnable or “available to execute” and once it begins execution it executes without waiting on any other OCR objects. In OCR, we package a task into an Event Driven Task or an EDT.

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*.

The OCR API defines the function prototype and return values expected by an EDT function. These include:

- The parameters of the EDT function which are copied by value when the EDT is created.
- Dynamic dependences expressed through a dependence array that is formed at runtime from explicit user-specified dependences.
- An optional GUID of a OCR object holding data (a *data block*) that will be used to satisfy the EDT’s post slot. This is the return value of the function.

When `ocrEdtCreate()` is used to create an EDT, it returns one or two GUIDs: 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 EDT⁵ When the OCR function returns a data block, the GUID of that data block is used to satisfy the 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()` API to trigger other OCR objects through explicitly created dependence links.

OCR defines one special type of EDT; the *finish EDT*. An EDT always executes 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 EDTs created within its 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.

1.3.2.2. 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

⁵It 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.

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 [7].

When an event is satisfied, it can optionally attach a data block to the 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*.

Events have one pre-slot except for latch-events which have two pre-slots.

1.3.3. Trigger rule

Events “trigger” when the appropriate *trigger rule* is met. The default trigger rule for events is when the link on their pre-slot is satisfied, the event triggers and passes the state from the pre-slot to its post slot. For example, if the pre-slot has an associated data block GUID, that data block GUID will be propagated through the event’s 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 is ignored.

1.3.3.1. 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. This increments the reference counter for the data block and plays a key role in managing the memory of an OCR program. 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 different 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). 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 benefit by defining 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 four modes:

1. *Read Only*: The EDT is stating that it will only read from the data block. This enables the runtime to provide a copy but with no need to manage the data blocks to support a subsequent step to “merge” updates upon release of the data block. Alternatively, the OCR runtime may choose to not schedule any other EDT that accesses the same read only DB in an *Intent to write* or *Exclusive write* mode. Note that a write to a read only data block may or may not become

visible to other EDTs (it is implementation dependent). No error will be flagged but the resulting state of the data block is undefined.

2. *Non-coherent read*: The EDT is stating that it will only read from the Data Block. The EDT 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 non-coherent read access.
3. *Intent to write* (default mode): The EDT is possibly going to write to the data block but does not require exclusive access to it. The programmer is responsible for synchronizing between EDTs that can potentially write to the same data block at the same time. Note that this theoretically permits the programmer to write a data race but also enables the programmer to write programs that update two “sections” of the same data block concurrently and in a race-free manner.
4. *Exclusive write*: The EDT requires that it is the only EDT writing to a data block at a given time. If multiple EDTs are runnable and want to access the same data block in *exclusive write* mode, the runtime will serialize the execution of these EDTs.

1.3.4. OCR program execution

An OCR computation starts as a single EDT called the `mainEDT()`. If a programmer does not provide a `main()` function, the OCR runtime system will create a `main()` function that sets up the OCR environment and calls the user provided `mainEDT()`. If a programmer chooses to provide his or her own `main()` function, then it is his or her responsibility to set up the OCR environment. The `mainEDT()` function has the function prototype:

```
1 #include <ocr.h>
   ocrGuid_t mainEdt(
       u32 paramc,           // number of parameters for mainEDT
       u64* paramv,         // array of parameters for mainEDT
       u32 depec,           // number of dependences for mainEDT
       ocrEdtDep_t depv[] ) // array of parameters for mainEDT
   {
       // Put the code for the mainEDT here
       ocrShutdown();      // shut down OCR once all resources have been released
11  return NULL_GUID;
   }
```

The details behind the parameters and dependences are described in Chapter 2.

The advantage of the letting the OCR runtime create the `main()` function is the programmer doesn't need to manage the low level details of initializing and cleanly shutting down OCR. This approach, however, does not work if the programmer wishes to use OCR inside a larger body of software perhaps as part of a library that must inter-operate with other runtimes. Hence, the OCR specification defines a set of functions to support this “library mode” of launching an OCR program.

In non-library mode (using just `mainEdt`), the arguments (`argc` and `argv`) are still communicated through the use of the first data block passed into `mainEdt`. The programmer can

use the functions `getArgc` and `getArgv` to gain access to these values. These functions are defined in Chapter 2.

Once the main EDT is launched, it builds a directed acyclic graph of OCR objects (such as other EDTs) with the post-slot of one OCR object connected to the pre-slots of subsequent OCR objects (through links). These links imply either explicit events or the implied event associated with the post-slot of an EDT. When an EDT either completes its task or when it wishes to signal another OCR object, it sets the associated event to “satisfied” which triggers the event to signal OCR objects connected to the link associated with the event.

Links can imply control dependences or, when a data block is associated with an event, they imply data flow between OCR objects. In either case, the events constrain the order of execution of EDTs typically executing the program as a data flow program.

With the overall structure of an OCR program’s DAG defined, we now turn to the behavior of a single EDT. An EDT waits until all of its pre-slots are satisfied. At that point the EDT is said to be *runnable*. 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 into 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* or *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-slot 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 + data” dependence. In the other case, no data-block will be made available and the dependence is akin to a pure control dependence.

DB to Event Adding a dependence between a data-block and an event is equivalent to satisfying the event with the data-block.

DB 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 dependences between OCR objects which are explicit to the runtime.

- When an EDT completes, it releases all resources associated with the EDT. It then satisfies the event implied by its post-slot and triggers the link to any objects connected to its post-slot. It can optionally pass a data block along with this event as the return value from the EDT function.

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.

1.4. 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).

To construct a memory model for OCR, we need to present a few definitions. The operations inside a task execute in a non-blocking manner. The order of such operations are defined by the *sequenced-before* relations 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 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 allow us to define the memory model for OCR.

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 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 can release the data block.

Any function in the OCR runtime that releases a data block must assure that all loads and stores to the data block occur before the data block is released and that the release must complete before the function returns.

The only way to establish a synchronized-with relation is through the behavior of events. If the pre-slot EDT2 is connected to the post-slot of EDT1, then EDT2 waits for 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 the two EDTs if we define the following rule for OCR.

An EDT must complete the release of all of its resources 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 relationship if we define the following rule.

If an EDT uses a DB to satisfy an event, all writes to that data block from the EDT must complete before the event is triggered.

Without this rule we can not 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 construct code where 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 results 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 race⁶ 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 to 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 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

⁶A *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

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.

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 8-byte aligned granularity. Overlapping writes to non-aligned or smaller than 8-byte granularity are not defined.

This 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.

1.5. Organization of this document

The remainder of this document is structured as follows:

- Chapter 2 contains the OCR reference manual generated from the OCR source code by the *Doxygen* package.
- Appendix A contains a set of pedagogical examples.
- Appendix B documents the “change history” for OCR and this specification.

2. OCR API Documentation

2.1. Module Index

2.1.1. Modules

Here is a list of all modules:

Data-block management for OCR	23
Event Driven Task API	32
Event Management for OCR	29
Limited "standard" API for OCR	36
OCR dependence APIs	35
OCR extensions/experimental APIs	17
Affinity extension	17
Interface for runtimes built on top of OCR	21
OCR used as a library extension	19
Support calls for OCR	49
Types and constants used in OCR	38
General types and constants	42
Types and constants associated with EDTs	45
Types and constants associated with data blocks	43
Types and constants associated with events	47

2.2. Data Structure Index

2.2.1. Data Structures

Here are the data structures with brief descriptions:

ocrConfig_t Data-structure containing the configuration parameters for the runtime	51
ocrEdtDep_t Type of values passed to an EDT on each pre-slot	52

2.3. Module Documentation

2.3.1. OCR extensions/experimental APIs

Tuning language API for OCR. This is an experimental feature.

Modules

- [Affinity extension](#)

This extension is primarily used for the distributed implementation of OCR to specify better placement for EDTs and data blocks.

- [Interface for runtimes built on top of OCR](#)

Contains additional APIs to enable other runtimes to be developed on top of OCR.

- [OCR used as a library extension](#)

OCR APIs to use OCR in a library mode.

2.3.1.1. Detailed Description

Tuning language API for OCR. This is an experimental feature. These APIs are experimental or extensions that are not necessarily supported on all platforms. They are subject to change

2.3.2. Affinity extension

This extension is primarily used for the distributed implementation of OCR to specify better placement for EDTs and data blocks.

Enumerations

- enum `ocrAffinityKind` { `AFFINITY_CURRENT`, `AFFINITY_PD`, `AFFINITY_PD_MASTER` }

Types of affinities.

Functions

- `u8 ocrAffinityCount (ocrAffinityKind kind, u64 *count)`

Returns a count of affinity GUIDs of a particular kind.

- **u8 ocrAffinityGet** (**ocrAffinityKind** kind, **u64** *count, **ocrGuid_t** *affinities)

Gets the affinity GUIDs of a particular kind. The 'affinities' array must have been previously allocated and big enough to contain 'count' GUIDs.

- **u8 ocrAffinityGetCurrent** (**ocrGuid_t** *affinity)

Returns an affinity the currently executing EDT is affinitized to.

2.3.2.1. Detailed Description

This extension is primarily used for the distributed implementation of OCR to specify better placement for EDTs and data blocks.

2.3.2.2. Enumeration Type Documentation

2.3.2.2.1. enum ocrAffinityKind Types of affinities.

You can query for the affinities corresponding to policy domains as well as for your own affinities

Enumerator

AFFINITY_CURRENT Affinities of the current EDT

AFFINITY_PD Affinities of the policy domains. You can then affinitize EDTs and data blocks to these affinities in the creation calls

AFFINITY_PD_MASTER Runtime reserved (do not use)

Definition at line 52 of file ocr-affinity.h.

2.3.2.3. Function Documentation

2.3.2.3.1. u8 ocrAffinityCount (**ocrAffinityKind** kind, **u64** * count) Returns a count of affinity GUIDs of a particular kind.

Parameters

in	kind	The affinity kind to query for. See ocrAffinityKind
out	count	Count of affinity GUIDs of that kind in the system

Returns

a status code

- 0: successful

2.3.2.3.2. u8 ocrAffinityGet (**ocrAffinityKind** kind, **u64** * count, **ocrGuid_t** * affinities) Gets the affinity GUIDs of a particular kind. The 'affinities' array must have been previously

allocated and big enough to contain 'count' GUIDs.

Parameters

in	<i>kind</i>	The affinity kind to query for. See ocrAffinityKind
in, out	<i>count</i>	As input, requested number of elements; as output the actual number of elements returned
out	<i>affinities</i>	Affinity GUID array

Returns

- a status code
- 0: successful

2.3.2.3.3. u8 ocrAffinityGetCurrent (ocrGuid_t * *affinity*) Returns an affinity the currently executing EDT is affinitized to.

An EDT may have multiple affinities. The programmer should rely on [ocrAffinityCount\(\)](#) and [ocrAffinityGet\(\)](#) to query all affinities.

Parameters

out	<i>affinity</i>	One affinity GUID for the currently executing EDT
-----	-----------------	---

Returns

- a status code
- 0: successful

2.3.3. OCR used as a library extension

OCR APIs to use OCR in a library mode.

Data Structures

- struct [ocrConfig_t](#)
Data-structure containing the configuration parameters for the runtime.
- void [ocrInit](#) ([ocrConfig_t](#) *ocrConfig)
Bring up the OCR runtime.
- void [ocrParseArgs](#) (int argc, const char *argv[], [ocrConfig_t](#) *ocrConfig)
Parses the arguments passed to main and extracts the relevant information to initialize OCR.
- **u8 ocrFinalize** ()
Prepares to tear down the OCR runtime.

- `ocrGuid_t ocrWait (ocrGuid_t outputEvent)`

Waits on the satisfaction of an event.

2.3.3.1. Detailed Description

OCR APIs to use OCR in a library mode. OCR is designed to be used in a standalone manner (using `mainEdt`) but can also be called from a traditional `main` function. These functions enable the programmer to startup and shutdown the OCR runtime

Describes the OCR as library API

When OCR is used as a library, the user need to explicitly bring up and tear down the runtime using the APIs provided here.

2.3.3.2. Function Documentation

2.3.3.2.1. `u8 ocrFinalize ()` Prepares to tear down the OCR runtime.

This call prepares the runtime to be torn-down. This call will only return after the OCR program completes (ie: after the program calls `ocrShutdown()`).

Returns

the status code of the OCR program:

- 0: clean shutdown, no errors
- non-zero: user provided error code to `ocrAbort()`

2.3.3.2.2. `void ocrInit (ocrConfig_t * ocrConfig)` Bring up the OCR runtime.

This function needs to be called to bring up the runtime. It should be called once for each runtime that needs to be brought up.

Parameters

in	<i>ocrConfig</i>	Configuration parameters to bring up the runtime
----	------------------	--

2.3.3.2.3. `void ocrParseArgs (int argc, const char * argv[], ocrConfig_t * ocrConfig)` Parses the arguments passed to `main` and extracts the relevant information to initialize OCR.

This should be called prior to `ocrInit()` to populate the `ocrConfig_t` variable needed by `ocrInit()`.

Parameters

in	<i>argc</i>	The number of elements in <i>argv</i>
in	<i>argv</i>	Array of <code>char * argumetns</code> .

in, out	<i>ocrConfig</i>	Pointer to an ocrConfig ocrParseArgs will populate. ocrConfig needs to have already been allocated
---------	------------------	--

2.3.3.2.4. ocrGuid_t ocrWait (ocrGuid_t *outputEvent*) Waits on the satisfaction of an event.

Warning

The event waited on must be a persistent event (in other words, not a LATCH or ONCE event). This call may be slow and inefficient. It is meant only to call OCR from sequential code and should be avoided in a pure OCR program

Parameters

in	<i>outputEvent</i>	GUID of the event to wait on
----	--------------------	------------------------------

Returns

The GUID of the data block on the post-slot of the waited-on event

2.3.4. Interface for runtimes built on top of OCR

Contains additional APIs to enable other runtimes to be developed on top of OCR.

- [ocrGuid_t ocrElsUserGet](#) (u8 offset)
 - Get the value stored at 'offset' in the current EDT's local storage.*
- void [ocrElsUserSet](#) (u8 offset, ocrGuid_t data)
 - Set the value stored at 'offset' in the current EDT's local storage.*
- [ocrGuid_t currentEdtUserGet](#) ()
 - Get the GUID of the calling EDT.*
- [u64 ocrNbWorkers](#) ()
 - Get the number of workers the runtime currently uses.*
- [ocrGuid_t ocrCurrentWorkerGuid](#) ()
 - Get the GUID of the calling worker.*
- [u8 ocrInformLegacyCodeBlocking](#) ()
 - Inform the OCR runtime that the currently executing thread is logically blocked.*

2.3.4.1. Detailed Description

Contains additional APIs to enable other runtimes to be developed on top of OCR. Defines additional API for runtime implementers

Warning

These APIs are not fully supported at this time and should be used with caution

2.3.4.2. Function Documentation

2.3.4.2.1. `ocrGuid_t currentEdtUserGet ()` Get the GUID of the calling EDT.

Returns

the GUID of the calling EDT or `NULL_GUID` if this code is running outside an EDT (runtime code)

2.3.4.2.2. `ocrGuid_t ocrCurrentWorkerGuid ()` Get the GUID of the calling worker.

Note

Exposed as a convenience to runtime implementors, may be deprecated anytime.

Returns

the GUID of the calling worker

2.3.4.2.3. `ocrGuid_t ocrElsUserGet (u8 offset)` Get the value stored at 'offset' in the current EDT's local storage.

Each EDT can store some information in its metadata in a way akin to TLS. This call retrieves this information from the ELS. The programmer can set this information using [ocrElsUserSet\(\)](#)

Parameters

<code>in</code>	<code>offset</code>	Offset (in bytes) in the ELS to fetch
-----------------	---------------------	---------------------------------------

Returns

the value at the requested offset or `NULL_GUID` if there is no ELS support

Warning

Must be called from within an EDT code.

2.3.4.2.4. `void ocrElsUserSet (u8 offset, ocrGuid_t data)` Set the value stored at 'offset' in the current EDT's local storage.

See Also

[ocrElsUserGet\(\)](#)

Parameters

in	<i>offset</i>	Offset (in bytes) in the ELS to set
in	<i>data</i>	Value to write at that offset

Note

This is a no-op if there is no ELS support

2.3.4.2.5. u8 ocrInformLegacyCodeBlocking () Inform the OCR runtime that the currently executing thread is logically blocked.

Note

Very experimental

2.3.4.2.6. u64 ocrNbWorkers () Get the number of workers the runtime currently uses.

Note

Exposed as a convenience to runtime implementors, may be deprecated anytime.

Returns

the number of workers

2.3.5. Data-block management for OCR

Data block management API for OCR.

Functions

- [u8 ocrDbCreate](#) ([ocrGuid_t](#) *db, void **addr, [u64](#) len, [u16](#) flags, [ocrGuid_t](#) affinity, [ocrInDbAllocator_t](#) allocator)

Request the creation of a data block.

- [u8 ocrDbDestroy](#) ([ocrGuid_t](#) db)

Request for the destruction of a data block.

- [u8 ocrDbRelease](#) ([ocrGuid_t](#) db)

Release the DB (indicates that the EDT no longer needs to access it)

- [u8 ocrDbMalloc](#) ([ocrGuid_t](#) guid, [u64](#) size, void **addr)

Allocates memory inside a data block in a way similar to malloc.

- **u8 ocrDbMallocOffset** (**ocrGuid_t** guid, **u64** size, **u64** *offset)

Allocates memory inside a data block in a way similar to malloc.

- **u8 ocrDbFree** (**ocrGuid_t** guid, void *addr)

Frees memory allocated through ocrDbMalloc()

- **u8 ocrDbFreeOffset** (**ocrGuid_t** guid, **u64** offset)

Frees memory allocated through ocrDbMallocOffset()

- **u8 ocrDbCopy** (**ocrGuid_t** destination, **u64** destinationOffset, **ocrGuid_t** source, **u64** sourceOffset, **u64** size, **u64** copyType, **ocrGuid_t** *completionEvt)

Copies data between two data blocks in an asynchronous manner.

2.3.5.1. Detailed Description

Data block management API for OCR. Describes the data block API for OCR

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)
- non-overlapping with other distinct data blocks
- 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 EDT, whichever comes first)

2.3.5.2. Function Documentation

2.3.5.2.1. **u8 ocrDbCopy** (**ocrGuid_t** destination, **u64** destinationOffset, **ocrGuid_t** source, **u64** sourceOffset, **u64** size, **u64** copyType, **ocrGuid_t** * completionEvt)

Copies data between two data blocks in an asynchronous manner.

This call will trigger the creation of an EDT which will perform a copy from a source data block into a destination data block. Once the copy is complete, the event with GUID 'completionEvt' will be satisfied. That event will carry the destination data block.

The type of GUID passed in as source also determines the starting point of the copy:

- if it is an event GUID, the EDT will be available to run when that event is satisfied. The data block carried by that event will be used as the source data block

- if it is a data block GUID, the EDT is immediately available to run and will be used as the source data block

Parameters

in	<i>destination</i>	Data block to copy to (must already be created and large enough to contain copy)
in	<i>destination-Offset</i>	Offset from the start of the destination data block to copy to (in bytes)
in	<i>source</i>	Data block to copy from
in	<i>sourceOffset</i>	Offset from the start of the destination data block to copy from (in bytes)
in	<i>size</i>	Number of bytes to copy
in	<i>copyType</i>	Reserved
out	<i>completionEvt</i>	GUID of the event that will be satisfied when the copy is successful

Returns

a status code

- 0: successful (note that this does not mean that the copy was done)
- EINVAL: Invalid values for one of the arguments
- EPERM: Overlapping data blocks
- ENOMEM: Destination too small to copy into or source too small to copy from

Note

This call is not supported at this time.

2.3.5.2.2. `u8 ocrDbCreate (ocrGuid_t * db, void ** addr, u64 len, u16 flags, ocrGuid_t affinity, ocrInDbAllocator_t allocator)` Request the creation of a data block.

On successful creation, the returned memory location will be 8 byte aligned. `ocrDbCreate` also implicitly acquires the data-block for the calling EDT unless `DB_PROP_NO_ACQUIRE` is specified

Parameters

out	<i>db</i>	On successful creation, contains the GUID for the newly created data block. Will be NULL if the call fails
out	<i>addr</i>	On successful creation, contains the 64 bit address if <code>DB_PROP_NO_ACQUIRE</code> is not specified. NULL otherwise
in	<i>len</i>	Size in bytes of the block to allocate.
in	<i>flags</i>	Flags to create the data block. Currently, the following properties are supported: <ul style="list-style-type: none"> • <code>DB_PROP_NONE</code>, indicates no specific properties • <code>DB_PROP_NO_ACQUIRE</code>, the DB will be created but not acquired (<code>addr</code> will be NULL).

<i>in, out</i>	<i>affinity</i>	GUID to indicate the affinity container of this DB. This is currently unsupported and NULL_GUID should be passed
<i>in</i>	<i>allocator</i>	Allocator to use to allocate within the data block. Supported values are given by ocrInDbAllocator_t

Returns

a status code:

- - : successful
- ENXIO : affinity is invalid
- ENOMEM: allocation failed because of insufficient memory
- EINVAL: invalid arguments (flags or something else)
- EBUSY : the agent that is needed to process this request is busy. Retry is possible.
- EPERM : trying to allocate in an area of memory that is not allowed

Note

The default allocator (NO_ALLOC) will disallow calls to `ocrDbMalloc` and `ocrDbFree`. If an allocator is used, part of the data block's space will be taken up by the allocator's management overhead

2.3.5.2.3. u8 ocrDbDestroy (ocrGuid_t db)

Request for the destruction of a data block.

The EDT does not need to have acquired the data block to destroy it. `ocrDbDestroy()` will request destruction of the DB but the DB will only be destroyed once all other EDTs that have acquired it release it (either implicitly at the end of the EDT or with an explicit `ocrDbRelease()` call).

Note

If the EDT has acquired this DB, this call implicitly releases the DB.

Once a data block has been marked as 'to-be-destroyed' by this call, the following operations on the same data block will result in an error:

- calling `ocrDbDestroy()` again (will return EPERM) The following operations will produce undefined behavior:
 - accessing the actual location of the data block (through a pointer)
 - acquiring the data block (implicit on EDT start)

Parameters

<i>in</i>	<i>db</i>	Data block to destroy
-----------	-----------	-----------------------

Returns

a status code:

- 0: successful
- EPERM: db was already destroyed

- EINVAL: db does not refer to a valid data-block

2.3.5.2.4. u8 ocrDbFree (ocrGuid_t *guid*, void * *addr*) Frees memory allocated through [ocrDbMalloc\(\)](#)

Parameters

in	<i>guid</i>	Data block to free from
in	<i>addr</i>	Address to free (as returned by ocrDbMalloc())

Returns

- a status code
 - 0: successful
 - EINVAL: Data block does not support allocation or *addr* is invalid

Warning

The address '*addr*' must have been allocated before the release of the containing data block. Use [ocrDbFreeOffset](#) if allocating and freeing across EDTs for example

Note

This call is not supported at this time.

2.3.5.2.5. u8 ocrDbFreeOffset (ocrGuid_t *guid*, u64 *offset*) Frees memory allocated through [ocrDbMallocOffset\(\)](#)

Parameters

in	<i>guid</i>	Data block to free from
in	<i>offset</i>	Offset to free (as returned by ocrDbMallocOffset())

Returns

- a status code
 - 0: successful
 - EINVAL: Data block does not support allocation or *offset* is invalid

Note

This call is not supported at this time.

2.3.5.2.6. u8 ocrDbMalloc (ocrGuid_t *guid*, u64 *size*, void ** *addr*) Allocates memory *inside* a data block in a way similar to malloc.

This will allocate a chunk of size '*size*' and return its address in '*addr*' using the memory available in the data block. This call, and others related to it, allow you to use a data block in a heap-like fashion

Parameters

in	<i>guid</i>	Data block to malloc from
in	<i>size</i>	Size of the chunk to allocate
out	<i>addr</i>	Address to the chunk allocated or NULL on failure

Returns

- a status code
- 0: successful
 - ENOMEM: Not enough space to allocate
 - EINVAL: Data block does not support allocation

Warning

The address returned is valid *only* for the current acquire of the data block (ie: it is an absolute address). Use [ocrDbMallocOffset\(\)](#) to get a more stable 'pointer'

Note

This call is not supported at this time.

2.3.5.2.7. u8 ocrDbMallocOffset (ocrGuid_t *guid*, u64 *size*, u64 * *offset*) Allocates memory *inside* a data block in a way similar to malloc.

This call is very similar to ocrDbMalloc except that it returns the location of the memory allocated as an *offset* from the start of the data block. This is a more preferred method as this allows the returned pointer to be used across EDTs (provided they all have access to the data block).

Parameters

in	<i>guid</i>	Data block to malloc from
in	<i>size</i>	Size of the chunk to allocate
out	<i>offset</i>	Offset of the chunk allocated in the data block

Returns

- a status code
- 0: successful
 - ENOMEM: Not enough space to allocate
 - EINVAL: Data block does not support allocation

Note

This call is not supported at this time.

2.3.5.2.8. u8 ocrDbRelease (ocrGuid_t *db*) Release the DB (indicates that the EDT no longer needs to access it)

This call can be used to indicate an early release of the data block (ie: it is not needed in the rest of the EDT). Once the data block is released, pointers that were previously associated with it are

invalid and should not be used to access the data.

The functionality of `ocrDbRelease()` is implicitly contained in:

- `ocrDbDestroy()`
- EDT exit

Note

`ocrDbRelease` should only be called *once* at most.

Parameters

<code>in</code>	<code>db</code>	Data block to release
-----------------	-----------------	-----------------------

Returns

a status code:

- 0: successful
- EINVAL: `db` does not refer to a valid data block
- EACCES: EDT has not acquired the data block and therefore cannot release it

2.3.6. Event Management for OCR

OCR APIs for EDTs, events and dependences.

Functions

- `u8 ocrEventCreate` (`ocrGuid_t *guid`, `ocrEventTypes_t eventType`, `bool takesArg`)

Creates 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 along to the event.

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

Satisfy the specified pre-slot of an event.

2.3.6.1. Detailed Description

OCR APIs for EDTs, events and dependences. Describes the event management APIs for OCR

2.3.6.2. Function Documentation

2.3.6.2.1. `u8 ocrEventCreate (ocrGuid_t * guid, ocrEventTypes_t eventType, bool takesArg)` Creates an event.

This function creates a new programmer-managed event identified by the returned GUID.

Parameters

out	<i>guid</i>	The GUID created by the runtime for the new event
in	<i>eventType</i>	The type of event to create. See ocrEventTypes_t
in	<i>takesArg</i>	True if this event will potentially carry a data block on satisfaction, false otherwise

Returns

- a status code
 - 0: successful
 - ENOMEM: If space cannot be found to allocate the event
 - EINVAL: If *eventType* was malformed or is incompatible with *takesArg*

2.3.6.2.2. `u8 ocrEventDestroy (ocrGuid_t guid)` Explicitly destroys an event.

Events such as ONCE or LATCH are automatically destroyed once they trigger; others, however, need to be explicitly destroyed by the programmer. This call enables this.

Parameters

in	<i>guid</i>	The GUID of the event to destroy
----	-------------	----------------------------------

Returns

- a status code
 - 0: successful
 - EINVAL: If *guid* does not refer to a valid event to destroy

2.3.6.2.3. `u8 ocrEventSatisfy (ocrGuid_t eventGuid, ocrGuid_t dataGuid)` Satisfy the first pre-slot of an event and optionally pass a data block along to the event.

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

- ONCE, IDEM and STICKY events will satisfy their post-slot upon satisfaction of their pre-slot
- LATCH events have a more complex rule; see [ocrEventTypes_t](#).

During satisfaction, the programmer may associate a data block to the pre-slot of the event. Depending on the event type, that data block will be passed along to its post-slot:

- ONCE, IDEM and STICKY events will pass along the data block to their post-slot

- LATCH events ignore any data block passed

Parameters

in	<i>eventGuid</i>	GUID of the event to satisfy
in	<i>dataGuid</i>	GUID of the data block to pass along or NULL_GUID

Returns

a status code

- 0: successful
- ENOMEM: If there is not enough memory. This is usually caused by a programmer error
- EINVAL: If the GUIDs do not refer to valid events/data blocks
- ENOPERM: If the event has already been satisfied or if the event does not take an argument and one is given or if the event takes an argument and none is given

An event satisfaction without the optional data block can be viewed as a pure control dependence whereas one with a data block is a control+data dependence

Note

On satisfaction, a ONCE event will pass the GUID of the optionally attached data block to all OCR objects waiting on it at that time and the event will destroy itself. IDEM and STICKY events will pass the GUID of the optionally attached data block to all OCR objects waiting on it at that time as well as any new objects linked to it until the event is destroyed.

2.3.6.2.4. `u8 ocrEventSatisfySlot (ocrGuid_t eventGuid, ocrGuid_t dataGuid, u32 slot)` Satisfy the specified pre-slot of an event.

This call is used primarily for LATCH events. `ocrEventSatisfySlot(eventGuid, dataGuid, 0)` is equivalent to `ocrEventSatisfy(eventGuid, dataGuid)`

See Also

[ocrEventSatisfy\(\)](#)

Parameters

in	<i>eventGuid</i>	GUID of the event to satisfy
in	<i>dataGuid</i>	GUID of the data block to pass along or NULL_GUID
in	<i>slot</i>	Pre-slot on the destination event to satisfy

Returns

a status code

- 0: successful
- ENOMEM: If there is not enough memory. This is usually caused by a programmer error
- EINVAL: If the GUIDs do not refer to valid events/data blocks
- ENOPERM: If the event has already been satisfied or if the event does not take an argument and one is given or if the event takes an argument and none is given

2.3.7. Event Driven Task API

APIs to manage EDTs in OCR.

Macros

- #define `ocrEdtTemplateCreate`(guid, funcPtr, paramc, depc) `ocrEdtTemplateCreate_internal`((guid), (funcPtr), (paramc), (depc), NULL)
Creates an EDT template.

Functions

- `u8 ocrEdtTemplateCreate_internal` (`ocrGuid_t *guid`, `ocrEdt_t funcPtr`, `u32 paramc`, `u32 depc`, `const char *funcName`)
Creates an EDT template.
- `u8 ocrEdtTemplateDestroy` (`ocrGuid_t guid`)
Destroy an EDT template.
- `u8 ocrEdtCreate` (`ocrGuid_t *guid`, `ocrGuid_t templateGuid`, `u32 paramc`, `u64 *paramv`, `u32 depc`, `ocrGuid_t *depv`, `u16 properties`, `ocrGuid_t affinity`, `ocrGuid_t *outputEvent`)
Creates an EDT instance from an EDT template.
- `u8 ocrEdtDestroy` (`ocrGuid_t guid`)
Destroy an EDT.

2.3.7.1. Detailed Description

APIs to manage EDTs in OCR.

2.3.7.2. Macro Definition Documentation

2.3.7.2.1. #define `ocrEdtTemplateCreate`(*guid*, *funcPtr*, *paramc*, *depc*) `ocrEdtTemplateCreate_internal`((guid**), (**funcPtr**), (**paramc**), (**depc**), **NULL**)** Creates an EDT template.

See Also

[ocrEdtTemplateCreate_internal\(\)](#)

Definition at line 174 of file ocr-edt.h.

2.3.7.3. Function Documentation

2.3.7.3.1. `u8 ocrEdtCreate (ocrGuid_t * guid, ocrGuid_t templateGuid, u32 paramc, u64 * paramv, u32 depc, ocrGuid_t * depv, u16 properties, ocrGuid_t affinity, ocrGuid_t * outputEvent)` Creates an EDT instance from an EDT template.

Parameters

out	<i>guid</i>	GUID of the newly created EDT type
in	<i>templateGuid</i>	GUID of the template to use to create this EDT
in	<i>paramc</i>	Number of parameters (64-bit values). Set to EDT_PARAM_DEF if you want to use the 'paramc' value specified at the time of the EDT template's creation
in	<i>paramv</i>	64-bit values for the parameters. This must be an array of paramc 64-bit values. These values are copied in. If paramc is 0, this must be NULL.
in	<i>depc</i>	Number of dependences for this EDT. Set to EDT_PARAM_DEF if you want to use the 'depc' value specified at the time of the EDT template's creation
in	<i>depv</i>	Values for the GUIDs of the dependences (if known). Note that all dependences added by this method will be in the DB_DEFAULT_MODE . Use ocrAddDependence() to add unknown dependences or dependences with another mode. This pointer should either be NULL or point to an array of size 'depc'.
in	<i>properties</i>	Used to indicate if this is a finish EDT (see EDT_PROP_FINISH). Use EDT_PROP_NONE as a default value.
in	<i>affinity</i>	Affinity container for this EDT. Can be NULL_GUID. This is currently an experimental feature
in, out	<i>outputEvent</i>	If not NULL on input, on successful return of this call, this will return the GUID of the event associated with the post-slot of the EDT. For a FINISH EDT, the post-slot of this event will be satisfied when the EDT and all of the child EDTs have completed execution. 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. If NULL, no event will be associated with the EDT's post-slot

Returns

- a status code
- 0: successful

2.3.7.3.2. `u8 ocrEdtDestroy (ocrGuid_t guid)` Destroy an EDT.

EDTs are normally destroyed after they execute. This call is provided if an EDT is created and the programmer later realizes that it will never become runnable.

Parameters

in	<i>guid</i>	GUID of the EDT to destroy
----	-------------	----------------------------

Returns

- a status code
- 0: successful

2.3.7.3.3. **u8 ocrEdtTemplateCreate_internal (ocrGuid_t * *guid*, ocrEdt_t *funcPtr*, u32 *paramc*, u32 *depc*, const char * *funcName*)** Creates an EDT template.

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; reusing the same template from multiple EDTs of the same type may improve performance as this allows the runtime to collect information about multiple instances of the same type of EDT.

Parameters

out	<i>guid</i>	Runtime created GUID for the newly created EDT Template
in	<i>funcPtr</i>	EDT function. This function must be of type ocrEdt_t .
in	<i>paramc</i>	Number of parameters that the EDTs will take. If not known or variable, this can be EDT_PARAM_UNK
in	<i>depc</i>	Number of pre-slots that the EDTs will have. If not known or variable, this can be EDT_PARAM_UNK
in	<i>funcName</i>	User-specified name for the template (used in debugging)

Returns

- a status code:
- 0: successful
 - ENOMEM: No memory available to allocate the template

Note

You should use [ocrEdtTemplateCreate\(\)](#) as opposed to this internal function. If OCR_ENABLE_EDT_NAMING is enabled, the C name of the function will be used as *funcName*.

2.3.7.3.4. **u8 ocrEdtTemplateDestroy (ocrGuid_t *guid*)** Destroy an EDT template.

This function can be called if no further EDTs will be created based on this template

Parameters

in	<i>guid</i>	GUID of the EDT template to destroy
----	-------------	-------------------------------------

Returns

- a status code

- 0: successful

2.3.8. OCR dependence APIs

APIs to manage OCR dependences.

Functions

- [u8 ocrAddDependence](#) ([ocrGuid_t](#) source, [ocrGuid_t](#) destination, [u32](#) slot, [ocrDbAccessMode_t](#) mode)

Adds a dependence between OCR objects:

2.3.8.1. Detailed Description

APIs to manage OCR dependences.

2.3.8.2. Function Documentation

2.3.8.2.1. u8 ocrAddDependence (ocrGuid_t source, ocrGuid_t destination, u32 slot, ocrDbAccessMode_t mode) Adds a dependence between OCR objects:

The following dependences can be added:

- Event to Event: The sink 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.
- Event to EDT : Upon satisfaction of the source event's post-slot, the pre-slot of the EDT will be satisfied. When the EDT runs, the data block associated with the post-slot of the event will be passed in through the depv array. This represents a control or a control+data dependence.
- DB to Event : `ocrAddDependence(db, evt, slot)` is equivalent to `ocrSatisfySlot(evt, db, slot)`
- DB 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 EDT runs, the data block will be passed in through the depv array.

Parameters

in	<i>source</i>	GUID of the source
in	<i>destination</i>	GUID of the destination
in	<i>slot</i>	Index of the pre-slot on the destination OCR object
in	<i>mode</i>	Access mode of the destination for the data block. See ocrDb-AccessMode_t .

Returns

- a status code
 - 0: successful
 - EINVAL: The slot number is invalid
 - ENOOPERM: The source and destination GUIDs cannot be linked with a dependence

2.3.9. Limited "standard" API for OCR

Limited "standard" API for OCR.

Macros

- #define `ASSERT(a)`
ASSERT macro to replace the assert functionality.
- #define `VERIFY(cond, format,...)`
Primitive method to print FAILURE or PASSED messages.

Functions

- `u32 PRINTF` (const char *fmt,...)
Console output.
- void `_ocrAssert` (bool val, const char *file, `u32` line)
Platform independent 'assert' functionality.

2.3.9.1. Detailed Description

Limited "standard" API for OCR. Describes the limited "standard" API for OCR

This file provides APIs for very limited standard functions that replicate certain libc functions.

Note

This may be extended in the future.

2.3.9.2. Macro Definition Documentation

2.3.9.2.1. #define ASSERT(a) ASSERT macro to replace the assert functionality.

Parameters

in	<i>a</i>	Condition for the assert
----	----------	--------------------------

Definition at line 75 of file ocr-std.h.

2.3.9.2.2. #define VERIFY(*cond*, *format*, ...) Value:

```
do {
    if(!(cond)) {
        PRINTF("FAILURE @ '%s:%d' " format, __FILE__, __LINE__, ## __VA_ARGS__); \
    } else {
        PRINTF("PASSED @ '%s:%d' " format, __FILE__, __LINE__, ## __VA_ARGS__); \
    }
} while(0);
```

Primitive method to print FAILURE or PASSED messages.

This macro is used to simply verify if the execution of the program is correct and prints out either a message starting with PASSED if successful or FAILURE if not

Parameters

in	<i>cond</i>	If true, will print a PASSED message.
in	<i>format</i>	Message to print (see PRINTF)
in	...	Arguments for message

Definition at line 91 of file ocr-std.h.

2.3.9.3. Function Documentation

2.3.9.3.1. void _ocrAssert (bool *val*, const char * *file*, u32 *line*) Platform independent 'assert' functionality.

This will cause the program to abort and return an assertion failure. This function should be called using the [ASSERT](#) macro

Parameters

in	<i>val</i>	If non-zero, will cause the assertion failure
in	<i>file</i>	File in which the assertion failure occurred
in	<i>line</i>	Line at which the assertion failure occurred

2.3.9.3.2. u32 PRINTF (const char * *fmt*, ...) Console output.

This maps to the typical C-like printf() functionality.

Parameters

in	<i>fmt</i>	Format specifier, as per printf(). Not all modifiers are supported. Standard ones like d, f, s, x, p work properly
in	...	Arguments, as per printf().

Returns

number of characters printed, as per printf().

2.3.10. Types and constants used in OCR

Basic types used throughout the OCR library.

Modules

- [General types and constants](#)
- [Types and constants associated with EDTs](#)
- [Types and constants associated with data blocks](#)
- [Types and constants associated with events](#)
- #define `OCR_EPERM` 1
Error codes for OCR.
- #define `OCR_ENOENT` 2
- #define `OCR_EINTR` 4
- #define `OCR_EIO` 5
- #define `OCR_ENXIO` 6
- #define `OCR_E2BIG` 7
- #define `OCR_ENOEXEC` 8
- #define `OCR_EAGAIN` 11
- #define `OCR_ENOMEM` 12
- #define `OCR_EACCES` 13
- #define `OCR_EFAULT` 14
- #define `OCR_EBUSY` 16
- #define `OCR_ENODEV` 19
- #define `OCR_EINVAL` 22
- #define `OCR_ENOSPC` 28
- #define `OCR_ESPIPE` 29
- #define `OCR_EROFS` 30
- #define `OCR_EDOM` 33

- #define `OCR_ERANGE` 34
- #define `OCR_ENOSYS` 38
- #define `OCR_ENOTSUP` 95
- #define `OCR_EGUIDEXISTS` 100
- #define `OCR_EACQ` 101
- #define `OCR_EPEND` 102
- #define `OCR_ECANCELED` 125

2.3.10.1. Detailed Description

Basic types used throughout the OCR library. Collection of types and constants used throughout the OCR API

2.3.10.2. Macro Definition Documentation

2.3.10.2.1. #define `OCR_E2BIG` 7 Argument list too long

Definition at line 25 of file ocr-errors.h.

2.3.10.2.2. #define `OCR_EACCES` 13 Permission denied

Definition at line 29 of file ocr-errors.h.

2.3.10.2.3. #define `OCR_EACQ` 101 Data block is already acquired

Definition at line 44 of file ocr-errors.h.

2.3.10.2.4. #define `OCR_EAGAIN` 11 Try again

Definition at line 27 of file ocr-errors.h.

2.3.10.2.5. #define `OCR_EBUSY` 16 Device or resource busy

Definition at line 31 of file ocr-errors.h.

2.3.10.2.6. #define `OCR_ECANCELED` 125 Operation canceled

Definition at line 46 of file ocr-errors.h.

2.3.10.2.7. #define OCR_EDOM 33 Math argument out of domain of func

Definition at line 37 of file ocr-errors.h.

2.3.10.2.8. #define OCR_EFAULT 14 Bad address

Definition at line 30 of file ocr-errors.h.

2.3.10.2.9. #define OCR_EGUIDEXISTS 100 The object referred to by the given GUID already exists

Definition at line 43 of file ocr-errors.h.

2.3.10.2.10. #define OCR_EINTR 4 Interrupted OCR runtime call

Definition at line 22 of file ocr-errors.h.

2.3.10.2.11. #define OCR_EINVAL 22 Invalid argument

Definition at line 33 of file ocr-errors.h.

2.3.10.2.12. #define OCR_EIO 5 I/O error

Definition at line 23 of file ocr-errors.h.

2.3.10.2.13. #define OCR_ENODEV 19 No such device

Definition at line 32 of file ocr-errors.h.

2.3.10.2.14. #define OCR_ENOENT 2 No such file or directory

Definition at line 21 of file ocr-errors.h.

2.3.10.2.15. #define OCR_ENOEXEC 8 Exec format error

Definition at line 26 of file ocr-errors.h.

2.3.10.2.16. #define OCR_ENOMEM 12 Out of memory

Definition at line 28 of file ocr-errors.h.

2.3.10.2.17. #define OCR_ENOSPC 28 No space left on device

Definition at line 34 of file ocr-errors.h.

2.3.10.2.18. #define OCR_ENOSYS 38 Function not implemented

Definition at line 39 of file ocr-errors.h.

2.3.10.2.19. #define OCR_ENOTSUP 95 Function is not supported

Definition at line 40 of file ocr-errors.h.

2.3.10.2.20. #define OCR_ENXIO 6 No such device or address

Definition at line 24 of file ocr-errors.h.

2.3.10.2.21. #define OCR_EPEND 102 Operation is pending

Definition at line 45 of file ocr-errors.h.

2.3.10.2.22. #define OCR_EPERM 1 Error codes for OCR.

Operation not permitted

Definition at line 20 of file ocr-errors.h.

2.3.10.2.23. #define OCR_ERANGE 34 Math result not representable

Definition at line 38 of file ocr-errors.h.

2.3.10.2.24. #define OCR_EROFS 30 Read-only file system

Definition at line 36 of file ocr-errors.h.

2.3.10.2.25. #define OCR_ESPIPE 29 Illegal seek

Definition at line 35 of file ocr-errors.h.

2.3.11. General types and constants

Macros

- #define **true** 1
- #define **TRUE** 1
- #define **false** 0
- #define **FALSE** 0
- #define **NULL_GUID** ((ocrGuid_t)0x0)
A NULL ocrGuid_t.
- #define **UNINITIALIZED_GUID** ((ocrGuid_t)-2)
An Uninitialized GUID (ie: never set)
- #define **ERROR_GUID** ((ocrGuid_t)-1)
An invalid GUID.

Typedefs

- typedef uint64_t **u64**
- typedef uint32_t **u32**
- typedef uint16_t **u16**
- typedef uint8_t **u8**
- typedef int64_t **s64**
- typedef int32_t **s32**
- typedef int8_t **s8**
- typedef **u8** **bool**
- typedef intptr_t **ocrGuid_t**

Type describing the unique identifier of most objects in OCR (EDTs, data-blocks, etc).

2.3.11.1. Detailed Description

2.3.11.2. Typedef Documentation

2.3.11.2.1. typedef intptr_t ocrGuid_t Type describing the unique identifier of most objects in OCR (EDTs, data-blocks, etc).

GUID type

Definition at line 54 of file ocr-types.h.

2.3.11.2.2. typedef int32_t s32 32-bit signed integer

Definition at line 33 of file ocr-types.h.

2.3.11.2.3. typedef int64_t s64 64-bit signed integer

Definition at line 32 of file ocr-types.h.

2.3.11.2.4. typedef int8_t s8 8-bit signed integer

Definition at line 34 of file ocr-types.h.

2.3.11.2.5. typedef uint16_t u16 16-bit unsigned integer

Definition at line 30 of file ocr-types.h.

2.3.11.2.6. typedef uint32_t u32 32-bit unsigned integer

Definition at line 29 of file ocr-types.h.

2.3.11.2.7. typedef uint64_t u64 64-bit unsigned integer

Definition at line 28 of file ocr-types.h.

2.3.11.2.8. typedef uint8_t u8 8-bit unsigned integer

Definition at line 31 of file ocr-types.h.

2.3.12. Types and constants associated with data blocks

Macros

- #define [DB_ACCESS_MODE_MASK](#) 0x1E
- #define [DB_DEFAULT_MODE](#) ((ocrDbAccessMode_t)DB_MODE_ITW)
- #define [DB_PROP_NONE](#) ((u16)0x0)
- #define [DB_PROP_NO_ACQUIRE](#) ((u16)0x10)
- #define [DB_PROP_SINGLE_ASSIGNMENT](#) ((u16)0x20)

Enumerations

- enum `ocrInDbAllocator_t` { `NO_ALLOC` = 0 }

Allocators that can be used to allocate within a data block.

- enum `ocrDbAccessMode_t` { `DB_MODE_RO` = 0x2, `DB_MODE_ITW` = 0x4, `DB_MODE_EW` = 0x8, `DB_MODE_NCR` = 0x10 }

Data block access modes.

2.3.12.1. Detailed Description

2.3.12.2. Macro Definition Documentation

2.3.12.2.1. `#define DB_ACCESS_MODE_MASK 0x1E` Runtime reserved constant

Definition at line 121 of file `ocr-types.h`.

2.3.12.2.2. `#define DB_DEFAULT_MODE ((ocrDbAccessMode_t)DB_MODE_ITW)`

Default access mode

Definition at line 122 of file `ocr-types.h`.

2.3.12.2.3. `#define DB_PROP_NO_ACQUIRE ((u16)0x10)` Property for a data block indicating that the data-block is just being created but does not need to be acquired at the same time (creation for another EDT)

Definition at line 125 of file `ocr-types.h`.

2.3.12.2.4. `#define DB_PROP_NONE ((u16)0x0)` Property for a data block indicating no special behavior

Definition at line 124 of file `ocr-types.h`.

2.3.12.2.5. `#define DB_PROP_SINGLE_ASSIGNMENT ((u16)0x20)` Property for a data block indicating single-assignment i.e. The user guarantees the data block is written once at creation time.

Note

This property is experimental and not implemented consistently.

Definition at line 133 of file `ocr-types.h`.

2.3.12.3. Enumeration Type Documentation

2.3.12.3.1. `enum ocrDbAccessMode_t` Data block access modes.

These are the modes with which an EDT can access a data block. OCR currently supports four modes:

- Read Only (RO): The EDT is stating that it will only read from the data block. In this mode, the runtime guarantees that the data block seen by the EDT is not modified by other concurrent EDTs (in other words, the data block does not change "under you". Any violation of the "no-write" contract by the program will result in undefined behavior (the write may or may not be visible to other EDTs depending on the implementation and specific runtime conditions).
- Non-coherent read (NCR): This mode is exactly the same as RO except that the runtime does not guarantee that the data block will not change.
- Intent to write (ITW) (default mode): The EDT is stating that it may or may not write to the data block. The user is responsible for synchronizing between EDTs that could potentially write to the same data block concurrently.
- Exclusive write (EW): The EDT requires that it be the only one accessing the data block. The runtime will not schedule any other EDT that accesses the same data block in EW or ITW mode concurrently. This can limit parallelism.

Enumerator

`DB_MODE_RO` Read-only mode
`DB_MODE_ITW` Intent-to-write mode (default mode)
`DB_MODE_EW` Exclusive write mode
`DB_MODE_NCR` Non-coherent read

Definition at line 114 of file `ocr-types.h`.

2.3.12.3.2. `enum ocrInDbAllocator_t` Allocators that can be used to allocate within a data block.

Data blocks can be used as heaps and allocators can be defined for these heaps. This enum lists the possible allocators.

Enumerator

`NO_ALLOC` No allocation is possible with the data block

Definition at line 86 of file `ocr-types.h`.

2.3.13. Types and constants associated with EDTs

Data Structures

- struct `ocrEdtDep_t`

Type of values passed to an EDT on each pre-slot.

Macros

- #define `EDT_PROP_NONE` ((u16) 0x0)
- #define `EDT_PROP_FINISH` ((u16) 0x1)
- #define `EDT_PARAM_UNK` ((u32)-1)

Constant indicating that the number of parameters or dependences to an EDT or EDT template is unknown.

- #define `EDT_PARAM_DEF` ((u32)-2)

Constant indicating that the number of parameters or dependences to an EDT is the same as the one specified in its template.

Typedefs

- typedef `ocrGuid_t(* ocrEdt_t)(u32 paramc, u64 *paramv, u32 depc, ocrEdtDep_t depv[])`

Type for an EDT.

2.3.13.1. Detailed Description

2.3.13.2. Macro Definition Documentation

2.3.13.2.1. #define EDT_PARAM_UNK ((u32)-1) Constant indicating that the number of parameters or dependences to an EDT or EDT template is unknown.

An EDT is created as an instance of an EDT template. The number of parameters or dependences for the EDT can either be specified when creating the template or when creating the EDT. This constant indicates that the number of parameters or dependences is still unknown (for example, when creating the template).

When the EDT is created, the number of parameters and dependences must be known (either specified in the template or the EDT). In other words, you cannot specify the number of parameters or dependences to be `EDT_PARAM_UNK` in both the creation of the template and the EDT.

Definition at line 190 of file `ocr-types.h`.

2.3.13.2.2. #define EDT_PROP_FINISH ((u16) 0x1) Property bits indicating a FINISH EDT

Definition at line 173 of file ocr-types.h.

2.3.13.2.3. #define EDT_PROP_NONE ((u16) 0x0) Property bits indicating a regular EDT

Definition at line 172 of file ocr-types.h.

2.3.13.3. Typedef Documentation

2.3.13.3.1. typedef ocrGuid_t(* ocrEdt_t)(u32 paramc, u64 *paramv, u32 depc, ocrEdtDep_t depv[]) Type for an EDT.

This is the function prototype for all EDTs.

Parameters

in	<i>paramc</i>	Number of non-data block parameters. A parameter is a 64-bit value known at the creation time of the EDT
in	<i>paramv</i>	Values of the 'paramc' parameters
in	<i>depc</i>	Number of dependences. This corresponds to the number of pre-slots for the EDT
in	<i>depv</i>	GUIDs and pointers to the data blocks passed to this EDT on its pre-slots. The GUID may be NULL_GUID if the pre-slot was a pure control dependence.

Returns

The GUID of a data block to pass along to the pre-slot of the output event optionally associated with this EDT. NULL_GUID can also be returned.

Definition at line 213 of file ocr-types.h.

2.3.14. Types and constants associated with events

Enumerations

- enum `ocrEventTypes_t` {
`OCR_EVENT_ONCE_T`, `OCR_EVENT_IDEM_T`, `OCR_EVENT_STICKY_T`,
`OCR_EVENT_LATCH_T`,
`OCR_EVENT_T_MAX` }

Types of OCR events.

- enum `ocrLatchEventSlot_t` { `OCR_EVENT_LATCH_DECR_SLOT` = 0,
`OCR_EVENT_LATCH_INCR_SLOT` = 1 }

Pre-slots for events.

2.3.14.1. Detailed Description

2.3.14.2. Enumeration Type Documentation

2.3.14.2.1. `enum ocrEventTypes_t` Types of OCR events.

Each OCR event has a type that is specified at creation. The type of the event determines its behavior, specifically:

- its persistency after it triggers its post-slot
- its trigger rule
- its behavior when satisfied multiple times

Enumerator

`OCR_EVENT_ONCE_T` A ONCE event simply passes along a satisfaction on its unique pre-slot to its post-slot. Once all OCR objects linked to its post-slot have been satisfied, the ONCE event is automatically destroyed.

`OCR_EVENT_IDEM_T` An IDEM event simply passes along a satisfaction on its unique pre-slot to its post-slot. The IDEM event persists until `ocrEventDestroy()` is explicitly called on it. It can only be satisfied once and subsequent satisfactions are ignored (use case: BFS, B&B..)

`OCR_EVENT_STICKY_T` A STICKY event is identical to an IDEM event except that multiple satisfactions result in an error

`OCR_EVENT_LATCH_T` A LATCH event has two pre-slots: a INCR and a DECR. Each slot is associated with an internal monotonically increasing counter that starts at 0. On each satisfaction of one of the pre-slots, the counter for that slot is incremented by 1. When both counters are equal (and non-zero), the post-slot of the latch event is triggered. Any data block passed along its pre-slots is ignored. A LATCH event has the same persistent as a ONCE event and is automatically destroyed when its post-slot is triggered.

`OCR_EVENT_T_MAX` This is *NOT* an event and is only used to count the number of event types. Its use is reserved for the runtime.

Definition at line 233 of file `ocr-types.h`.

2.3.14.2.2. `enum ocrLatchEventSlot_t` Pre-slots for events.

Currently, only the LATCH event has more than one pre-slot.

Enumerator

`OCR_EVENT_LATCH_DECR_SLOT` The decrement slot of a LATCH event

`OCR_EVENT_LATCH_INCR_SLOT` The increment slot of a LATCH event

Definition at line 267 of file ocr-types.h.

2.3.15. Support calls for OCR

Top level OCR library file. Include this file in any program making use of OCR. You do not need to include any other files unless using extended or experimental features present in the extensions/ folder.

Functions

- void `ocrShutdown` ()

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

- void `ocrAbort` (u8 errorCode)

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

- u64 `getArgc` (void *dbPtr)

Retrieves the traditional 'argc' value in mainEdt.

- char * `getArgv` (void *dbPtr, u64 count)

Gets the argument 'count' from the data-block containing the arguments.

2.3.15.1. Detailed Description

Top level OCR library file. Include this file in any program making use of OCR. You do not need to include any other files unless using extended or experimental features present in the extensions/ folder. Describes general support functions for OCR

2.3.15.2. Function Documentation

2.3.15.2.1. u64 `getArgc` (void * *dbPtr*)

Retrieves the traditional 'argc' value in mainEdt.

When starting, the first EDT (called mainEdt) gets a single data block that contains all of the arguments passed to the program. These arguments are packed in the following format:

- first 8 bytes: argc
- argc count of 8 byte values indicating the offsets from the start of the data block to the argument (ie: the first 8 byte value indicates the offset in bytes to the first argument)
- the arguments (NULL terminated character arrays)

This call will extract the number of arguments (argc)

Parameters

<i>dbPtr</i>	Pointer to the start of the argument data block
--------------	---

Returns

Number of arguments

2.3.15.2.2. char* getArgv (void * *dbPtr*, u64 *count*) Gets the argument 'count' from the data-block containing the arguments.

See Also

[getArgc\(\)](#) for an explanation

Parameters

<i>dbPtr</i>	Pointer to the start of the argument data-block
<i>count</i>	Index of the argument to extract

Returns

A NULL terminated string

2.3.15.2.3. void ocrAbort (u8 *errorCode*) Called by an EDT to indicate an abnormal end of an OCR program.

This call will cause the OCR runtime to shutdown with an error code. Calling this with 0 as an argument is equivalent to [ocrShutdown\(\)](#).

Parameters

<i>errorCode</i>	User defined error code returned to the environment
------------------	---

2.3.15.2.4. void ocrShutdown () Called by an EDT to indicate the end of an OCR program.

This call will cause the OCR runtime to shutdown

Note

If using the extended [ocrWait\(\)](#) call present in ocr-legacy.h, you do not need to use [ocrShutdown\(\)](#) as the termination of the OCR portion will be captured in the finish EDT's return

2.4. Data Structure Documentation

2.4.1. ocrConfig_t Struct Reference

Data-structure containing the configuration parameters for the runtime.

```
#include <ocr-lib.h>
```

Data Fields

- int [userArgc](#)
- char ** [userArgv](#)
- const char * [iniFile](#)

2.4.1.1. Detailed Description

Data-structure containing the configuration parameters for the runtime.

The contents of this struct can be filled in by calling [ocrParseArgs\(\)](#) or by setting them manually. The former method is strongly recommended.

Definition at line 49 of file ocr-lib.h.

2.4.1.2. Field Documentation

2.4.1.2.1. const char* ocrConfig_t::iniFile INI configuration file for the runtime

Definition at line 52 of file ocr-lib.h.

2.4.1.2.2. int ocrConfig_t::userArgc Application argc (after having stripped the OCR arguments)

Definition at line 50 of file ocr-lib.h.

2.4.1.2.3. char ocrConfig_t::userArgv** Application argv (after having stripped the OCR arguments)

Definition at line 51 of file ocr-lib.h.

The documentation for this struct was generated from the following file:

- ocr-lib.h

2.4.2. ocrEdtDep_t Struct Reference

Type of values passed to an EDT on each pre-slot.

```
#include <ocr-types.h>
```

Data Fields

- [ocrGuid_t guid](#)
- `void * ptr`

2.4.2.1. Detailed Description

Type of values passed to an EDT on each pre-slot.

An EDT with N pre-slots will receive an array of N elements of this type (its input dependences). Each dependence has the GUID of the data block passed along that pre-slot as well as a pointer to the data in the data block.

Note

The GUID passed to the EDT is *not* the GUID of the event linked to the pre-slot of the EDT but rather the GUID of the data block that was associated with that event. If no data block was associated, NULL_GUID is passed.

Definition at line 166 of file ocr-types.h.

2.4.2.2. Field Documentation

2.4.2.2.1. ocrGuid_t ocrEdtDep_t::guid GUID of the data block or NULL_GUID

Definition at line 167 of file ocr-types.h.

2.4.2.2.2. void* ocrEdtDep_t::ptr Pointer allowing access to the data block or NULL

Definition at line 168 of file ocr-types.h.

The documentation for this struct was generated from the following file:

- ocr-types.h

Bibliography

- [1] S. Chatterjee. *Runtime Systems for Extreme Scale Platforms*. PhD thesis, Rice University, Dec 2013. 6
- [2] 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. 5
- [3] 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
- [4] Y. Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010. 5
- [5] 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. 6
- [6] 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. 5
- [7] 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
- [8] V. Sarkar et al. DARPA Exascale Software Study report, September 2009. 2
- [9] V. Sarkar, W. Harrod, and A. E. Snaveley. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale. 2
- [10] D. Sbirlea, 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. 5

- [11] 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
- [12] 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. 5
- [13] 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
- [14] S. Taşirlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011. 5
- [15] S. Tasirlar. Scheduling Macro-Dataflow Programs on Task-Parallel Runtime Systems, Apr 2011. 5
- [16] 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
- [17] N. Vrvilo. Asynchronous Checkpoint/Restart for the Concurrent Collections Model, Aug 2014. MS thesis. 1
- [18] 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. 5

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>
3 ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("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` function¹.

¹Note that if the programmer *does* provide a `main` function, it is the responsibility of the programmer to properly initialize the runtime, call the first EDT to execute and properly shutdown the runtime. This method is detailed in `TODO` and is not recommended as it is not platform portable.

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

```

3  /* Example of a "fork-join" pattern in OCR
   *
   * Implements the following dependence graph:
   *
   *   mainEdt
   *  /      \
   * fun1    fun2
   *  \      /
   * shutdownEdt
   *
   */
13 #include "ocr.h"

ocrGuid_t fun1(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
18     int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid, (void **) &k, sizeof(int), 0, NULL_GUID, NO_ALLOC);
    k[0]=1;
    PRINTF("Hello from fun1, sending k = %lu\n", *k);
    return db_guid;
23 }

ocrGuid_t fun2(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
28     int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid, (void **) &k, sizeof(int), 0, NULL_GUID, NO_ALLOC);
    k[0]=2;
    PRINTF("Hello from fun2, sending k = %lu\n", *k);
    return db_guid;
33 }

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
38     PRINTF("Hello from shutdownEdt\n");
    int* data1 = (int*) depv[0].ptr;
    int* data2 = (int*) depv[1].ptr;
    PRINTF("Received data1 = %lu, data2 = %lu\n", *data1, *data2);
    ocrDbDestroy(depv[0].guid);
    ocrDbDestroy(depv[1].guid);
    ocrShutdown();
    return NULL_GUID;
43 }

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("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_GUID, &outputEvent1);
ocrEdtCreate(&edt2 , edt2_template , EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
EDT_PROP_NONE, NULL_GUID, &outputEvent2);
ocrEdtCreate(&edt3 , edt3_template , EDT_PARAM_DEF, NULL, 2, NULL, EDT_PROP_NONE,
58 NULL_GUID, NULL);

//Setup dependences for the shutdown EDT
ocrAddDependence(outputEvent1 , edt3 , 0, DB_MODE_RO);
ocrAddDependence(outputEvent2 , edt3 , 1, DB_MODE_RO);
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 EDT, 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. For the EDTs, `edt1`, `edt2` and `edt3`, the EDT functions are, `fun1`, `fun2` and `shutdownEdt`, respectively. 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` respectively. 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 dependencies 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 dependencies, they execute the associated EDT functions in parallel. These functions (`fun1` and `fun2`) creates 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.


```

43  ASSERT(paramc == 1);
    // paramv[0] is 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
48  ocrGuid_t dbGuid = NULL_GUID, stage2aTemplateGuid = NULL_GUID,
    stage2aEdtGuid = NULL_GUID;
    u64* dbPtr = NULL;
    ocrDbCreate(&dbGuid, (void*)&dbPtr, sizeof(u64), 0, NULL_GUID, NO_ALLOC);
    *dbPtr = IULL;

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

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

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

68  // 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_GUID, NO_ALLOC);
    *dbPtr = 2ULL;

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

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

    u64 *dbPtr = (u64*)depv[0].ptr;
83  ASSERT(*dbPtr == IULL); // We got this from stage1a

    *dbPtr = 3ULL; // Update the value

    // Pass the modified data-block to shutdown
88  ocrEventSatisfy((ocrGuid_t)paramv[0], depv[0].guid);

    return NULL_GUID;
}

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

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

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

103  *dbPtr = 4ULL; // Update the value

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

```

```

108
ocrGuid_t mainEdt(u32 paramc , u64* paramv , u32 depc , ocrEdtDep_t depv[]) {
    // Create the shutdown EDT
113 ocrGuid_t stage1aTemplateGuid = NULL_GUID, stage1bTemplateGuid = NULL_GUID,
    stage2bTemplateGuid = NULL_GUID, shutdownEdtTemplateGuid = NULL_GUID;
ocrGuid_t shutdownEdtGuid = NULL_GUID, stage1aEdtGuid = NULL_GUID,
    stage1bEdtGuid = NULL_GUID, stage2bEdtGuid = NULL_GUID,
118 evtGuid = NULL_GUID, stage1aOut = NULL_GUID, stage1bOut = NULL_GUID,
    stage2bOut = NULL_GUID;

ocrEdtTemplateCreate(&shutdownEdtTemplateGuid , shutdownEdt , 0, 2);
ocrEdtCreate(&shutdownEdtGuid , shutdownEdtTemplateGuid , 0, NULL, EDT_PARAM_DEF, NULL,
    EDT_PROP_NONE, NULL_GUID, NULL);

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

128 // 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 , 1, 1);
ocrEdtCreate(&stage1aEdtGuid , stage1aTemplateGuid , EDT_PARAM_DEF, &evtGuid ,
133 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_GUID, &stage1aOut);

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

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

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

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

// Event satisfied by 2a -> shutdown
ocrAddDependence(evtGuid , shutdownEdtGuid , 0, DB_DEFAULT_MODE);
// 2b -> shutdown
153 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);
158
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 it is the immediate dominator 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 145. 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 156. 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 runnable. 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 145 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 57;
- Through an output event as shown on Line 61. 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 88).

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.

```
/* Example usage of Finish EDT in FFT.
 *
5 * Implements the following dependence graph:
 *
 * MainEdt
 * |
 *
10 * FinishEdt
 * {
 *     DFT
```

```

15 *      / \
*      FFT-odd FFT-even
*      \ /
*      Twiddle
* }
* |
* Shutdown
20 *
*/

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

25 #define N      256
#define BLOCK_SIZE 16

// The below function performs a twiddle operation on an array x_in
30 // 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) {
35         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;
40         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);
45             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 +
50                 (twiddle_real*xr - twiddle_imag*xi);
            X_imag[k] = t_imag +

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

                (twiddle_imag*xr + twiddle_real*xi);
        }
60 }

// 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
65 // 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];
70     double *data = (double*)depv[0].ptr;
    ocrGuid_t dataGuid = depv[0].guid;
    u64 size = paramv[2];
    u64 step = paramv[3];
    u64 offset = paramv[4];
75     u64 step_offset = paramv[5];
    u64 blockSize = paramv[6];
    double *x_in = (double*)data;

```

```

80  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 {
85      // 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 };

90      ocrGuid_t edtGuid, edtGuid2, twiddleEdtGuid, finishEventGuid, finishEventGuid2;

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

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

        ocrAddDependence(dataGuid, edtGuid, 0, DB_MODE_ITW);
        ocrAddDependence(dataGuid, edtGuid2, 0, DB_MODE_ITW);
105     }

    return NULL_GUID;
}

110 // 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];
115    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);

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

    return NULL_GUID;
}

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

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

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

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

    ocrGuid_t dataGuid;

```

```

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

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

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

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

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

165 ocrAddDependence(dataGuid , edtGuid , 0 , DB_MODE_ITW);
ocrAddDependence(eventGuid , endGuid , 0 , DB_MODE_ITW);

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 67 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 111 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 gradually larger data sizes till the entire array has undergone the operation. When this is available, a final EDT termed `endEdt()` in Line 125 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 datablock created in Line 143. 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 datablock 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 94,97 and 101. Finally, the single `endEdt ()` instance in Line 157 is called only after all the EDTs spawned by the parent `fftComputeEdt ()` in Line 162, return.

A.5. Accessing a DataBlock with “Intent-To-Write” Mode

This example illustrates the usage model for datablocks accessed with the `Intent-To-Write (ITW)` mode. The `ITW` mode ensures that only one master copy of the datablock exists at any time inside a shared address space. Parallel EDTs can concurrently access a datablock 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 datablock with the `ITW` mode.

A.5.1. Code example

```

2  /* Example usage of ITW (Intent-To-Write)
   * datablock access mode in OCR
   *
   * Implements the following dependence graph:
   *
   *      mainEdt
   *      [ DB ]
   *      /      \
   * (ITW)/        \ (ITW)
   * /            \
   * EDT1          EDT2
   * \            /
   *      [ DB ]
   *      shutdownEdt
   *
   */
17
#include "ocr.h"
#define N 1000
22 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;
27
    for (i = lb; i < ub; i++)
        dbPtr[i] += i;

    return NULL_GUID;

```

```

32 }
ocrGuid_t awaitingEdt(u32 paramc , u64* paramv , u32 depc , ocrEdtDep_t depv[]) {
    u64 i;
    PRINTF("Done!\n");
37    u32 *dbPtr = (u32*)depv[0].ptr;
    for (i = 0; i < N; i++) {
        if (dbPtr[i] != i * 2)
            break;
    }
42
    if (i == N) {
        PRINTF("Passed Verification\n");
    } else {
        PRINTF("!!! FAILED !!! Verification\n");
47    }

    ocrDbDestroy(depv[0].guid);
    ocrShutdown();
    return NULL_GUID;
52 }

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

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

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

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

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

82    // 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_GUID, NULL);
87    ocrAddDependence(dbGuid , awaitingEdtGuid , 0, DB_MODE_RO);
    ocrAddDependence(exampleEventGuid1 , awaitingEdtGuid , 1, DB_DEFAULT_MODE);
    ocrAddDependence(exampleEventGuid2 , awaitingEdtGuid , 2, DB_DEFAULT_MODE);

    // START
92    PRINTF("Start!\n");
    ocrAddDependence(dbGuid , exampleEdtGuid1 , 0, DB_MODE_ITW);
    ocrAddDependence(dbGuid , exampleEdtGuid2 , 0, DB_MODE_ITW);

```

97

```
return NULL_GUID;
```

A.5.1.1. Details

The `mainEdt` creates a datablock (`dbGuid`) that may be concurrently updated by two children EDTs (`exampleEdtGuid1` and `exampleEdtGuid2`) using the ITW mode. `exampleEdtGuid1` and `exampleEdtGuid2` are each created with one dependence on each of them, 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 datablock `dbGuid`.

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

A.6. Accessing a DataBlock with “Exclusive-Write” Mode

The `Exclusive-Write` (EW) mode allows for an easy implementation of mutual exclusion of EDTs execution. When an EDT depend on one or several DBs in EW mode, the runtime guarantees it is the only EDT through the system to currently writing to those DBs. Hence, the EW mode is useful when one wants to guarantee there's no race condition writing to a Data-Block or when ordering among EDTs do not matter for as long as the execution is in mutual exclusion. The following examples shows how two EDTs may share access to a DB in ITW mode, while one EDT requires EW access. In this situation the programmer cannot assume in which order 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

```

2  /* Example usage of EW (Exclusive-Write)
   * datablock access mode in OCR
   *
   * Implements the following dependence graph:
   *
   *      mainEdt
   *      [ DB ]
   *      /  |  \
   * (ITW)/  |(ITW) \ (EW)
   *      /  |  \
   *   EDT1 EDT2 EDT3
   *      \  |  /
   *      \  |  /
   *      [ DB ]
   *      shutdownEdt
17  */
   #include "ocr.h"
22  #define NB_ELEM_DB 20

   ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
       // The fourth slot (3 as it is 0-indexed) was the DB.
       u64 * data = (u64 *) depv[3].ptr;
       u32 i = 0;
       while (i < NB_ELEM_DB) {
           PRINTF("%d ", data[i]);
           i++;
       }
       PRINTF("\n");
       // Destroying the DB implicitly releases it.
       ocrDbDestroy(depv[3].guid);
       // Instruct the runtime the application is done executing
       ocrShutdown();
       return NULL_GUID;
37  }

   ocrGuid_t writerEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
       // An EDT has access to the parameters and dependences it has been created with.
       // ocrEdtDep_t allow to access both the '.guid' of the dependence and the '.ptr'
       // Note that when an EDT has an event as a dependence and this event is satisfied
       // with a DB GUID, the .guid field contains the DB GUID, not the event GUID.
       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++;
       }
       // The GUID the output event of this EDT is satisfied with
       return NULL_GUID;
57  }

   ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
       void * dbPtr;
       ocrGuid_t dbGuid;
       u32 nbElem = NB_ELEM_DB;
       // Create a DataBlock (DB). Note that the currently executing EDT
       // acquires the DB in Intent-To-Write (ITW) mode
       ocrDbCreate(&dbGuid, &dbPtr, sizeof(u64) * NB_ELEM_DB, 0, NULL_GUID, NO_ALLOC);
62  }

```



```

67  u64 i = 0;
    int * data = (int *) dbPtr;
    while (i < nbElem) {
        data[i] = 0;
        i++;
    }
72  // Indicate to the runtime the current EDT is not using the DB anymore
    ocrDbRelease(dbGuid);

    // Create the sink EDT template. The sink EDT is responsible for
    // shutting down the application.
    // It has 4 dependences: EDT1, EDT2, EDT3 and the DB
77  ocrGuid_t shutdownEdtTemplateGuid;
    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0, 4);
    ocrGuid_t shutdownGuid;
    // Create the shutdown EDT indicating this instance of the shutdown EDT template
    // has the same number of parameters and dependences the template declares.
82  ocrEdtCreate(&shutdownGuid, shutdownEdtTemplateGuid, 0, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, NULL);
    // EDT is created, but it does not have its four dependences set up yet.
    // Set the third dependence of EDT shutdown to be the DB
    ocrAddDependence(dbGuid, shutdownGuid, 3, DB_MODE_RO);

87  // Writer EDTs have 3 parameters and 2 dependences
    ocrGuid_t writeEdtTemplateGuid;
    ocrEdtTemplateCreate(&writeEdtTemplateGuid, writerEdt, 3, 2);

92  // Create the event that enable EDT1, EDT2, EDT3 to run
    // It is a once event automatically destroyed when satisfy
    // has been called on it. Because of that we need to make
    // sure that all its dependences are set up before satisfied
    // is called.
97  ocrGuid_t eventStartGuid;
    ocrEventCreate(&eventStartGuid, OCR_EVENT_ONCE_T, false);

    // ITW '1' from 0 to N/2 (potentially concurrent with writer 1, but different range)
    ocrGuid_t oeWriter1Guid;
    ocrGuid_t writer1Guid;
102  // parameters composed of lower bound, upper bound, value to write.
    // parameters are passed by copy to the EDT, so it's ok to use the stack here.
    u64 writerParamv1[3] = {0, NB_ELEM_DB/2, 1};
    // Create the EDT1. Note the output event parameter 'oeWriter1Guid'.
107  // The output event is satisfied automatically by the runtime when EDT1
    // is done executing. This event is by default a ONCE event. The user must
    // make sure dependences on that event are set up before the EDT is scheduled.
    // This is the main reason why we have a start event. It allows to create all the
    // EDT before-hand and set up all the dependences before any scheduling occurs.
112  ocrEdtCreate(&writer1Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv1,
        EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &oeWriter1Guid);
    // Set up the sink EDT dependence slot 0 (2 dependences added so far)
    ocrAddDependence(oeWriter1Guid, shutdownGuid, 0, false);
    // EDT1 depends on the DB in ITW mode on its slot '0'
117  ocrAddDependence(dbGuid, writer1Guid, 0, DB_MODE_ITW);
    // EDT1 depends on the start event on its slot '1'
    ocrAddDependence(eventStartGuid, writer1Guid, 1, DB_MODE_RO);
    // At this point EDT

122  // ITW '2' from N/2 to N (potentially concurrent with writer 0, but different range)
    ocrGuid_t oeWriter2Guid;
    ocrGuid_t writer2Guid;
    u64 writerParamv2[3] = {NB_ELEM_DB/2, NB_ELEM_DB, 2};
    ocrEdtCreate(&writer2Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv2,
        EDT_PARAM_DEF, NULL,
127  EDT_PROP_NONE, NULL_GUID, &oeWriter2Guid);

```

```

132 // Set up the sink EDT dependence slot 1 (3 dependences added so far)
ocrAddDependence(oeWriter2Guid, shutdownGuid, 1, false);
ocrAddDependence(dbGuid, writer2Guid, 0, DB_MODE_ITW);
ocrAddDependence(eventStartGuid, writer2Guid, 1, DB_MODE_RO);

137 // EW '3' from N/4 to 3N/4
ocrGuid_t oeWriter3Guid;
ocrGuid_t writer3Guid;
u64 writerParamv3[3] = {NB_ELEM_DB/4, (NB_ELEM_DB/4)*3, 3};
ocrEdtCreate(&writer3Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv3,
142 EDT_PARAM_DEF, NULL,
EDT_PROP_NONE, NULL_GUID, &oeWriter3Guid);
// Set up the sink EDT dependence slot 2.
// At this point the shutdown EDT has all its dependences
// and will be eligible for scheduling when they are satisfied
ocrAddDependence(oeWriter3Guid, shutdownGuid, 2, false);
147 // EDT3 request the DB in Exclusive-Write (EW) mode. This is essentially
// introducing an implicit ordering dependence between all other EDTs
// that are also acquiring this DB. The actual EDTs execution ordering
// is schedule dependent.
ocrAddDependence(dbGuid, writer3Guid, 0, DB_MODE_EW);
ocrAddDependence(eventStartGuid, writer3Guid, 1, DB_MODE_RO);

152 // At this point all writers EDTs have their DB dependence satisfied and
// are only missing the start event to be satisfied.
// Doing so enable EDT1, EDT2, EDT3 to be eligible for scheduling.
// Because there's no control dependence among the writer EDT, the
// runtime is free to schedule them in any order, potentially in parallel.
// In this particular example, EDT1 and EDT2 can execute in parallel because
157 // they both access the DB in ITW mode which allow for concurrent writers.
// EDT3 will be executed in mutual exclusion with EDT1 and EDT2 at any point in time
// since it requires EW access.
ocrEventSatisfy(eventStartGuid, NULL_GUID);

162 return NULL_GUID;
}

```

A.6.1.1. Details

A.7. Acquiring contents of a DataBlock as a dependence input

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

A.7.1. Code example

```
3  /* 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
   *
   */
18 #include "ocr.h"
18 #define VAL 42
23 ocrGuid_t exampleEdt(u32 paramc, u64* paramv, u32 depec, ocrEdtDep_t depv[]) {
   ocrGuid_t *dbPtr = (ocrGuid_t*)depv[0].ptr;
   ocrGuid_t passedDb = dbPtr[0];
   PRINTF("Passing DB: 0x%x\n", passedDb);
   ocrDbDestroy(depv[0].guid);
   return passedDb;
28 }
33 ocrGuid_t awaitingEdt(u32 paramc, u64* paramv, u32 depec, ocrEdtDep_t depv[]) {
   u32 *dbPtr = (u32*)depv[0].ptr;
   PRINTF("Received: %u\n", dbPtr[0]);
   ocrDbDestroy(depv[0].guid);
   ocrShutdown();
   return NULL_GUID;
38 }
43 ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depec, ocrEdtDep_t depv[]) {
   // Create DBs
   u32* ptr0;
   ocrGuid_t* ptr1;
   ocrGuid_t db0Guid, db1Guid;
   ocrDbCreate(&db0Guid, (void*)&ptr0, sizeof(u32), DB_PROP_NONE, NULL_GUID, NO_ALLOC);
   ocrDbCreate(&db1Guid, (void*)&ptr1, sizeof(ocrGuid_t), DB_PROP_NONE, NULL_GUID,
48   NO_ALLOC);
   ptr0[0] = VAL;
   ptr1[0] = db0Guid;
   PRINTF("Sending: %u in DB: 0x%x\n", ptr0[0], db0Guid);
   ocrDbRelease(db0Guid);
   ocrDbRelease(db1Guid);
53 // Create Middle EDT
   ocrGuid_t exampleTemplGuid, exampleEdtGuid, exampleEventGuid;
   ocrEdtTemplateCreate(&exampleTemplGuid, exampleEdt, 0 /*paramc*/, 1 /*depc*/);
   ocrEdtCreate(&exampleEdtGuid, exampleTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
   EDT_PROP_NONE, NULL_GUID, &exampleEventGuid);
58 // Create AWAIT EDT
   ocrGuid_t awaitingTemplGuid, awaitingEdtGuid;
   ocrEdtTemplateCreate(&awaitingTemplGuid, awaitingEdt, 0 /*paramc*/, 1 /*depc*/);
   ocrEdtCreate(&awaitingEdtGuid, awaitingTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
   NULL,
   EDT_PROP_NONE, NULL_GUID, NULL);
```

```
63   ocrAddDependence(exampleEventGuid, awaitingEdtGuid, 0, DB_DEFAULT_MODE);  
    // START Middle EDT  
    ocrAddDependence(db1Guid, exampleEdtGuid, 0, DB_DEFAULT_MODE);  
    return NULL_GUID;  
}
```

A.7.1.1. Details

The mainEdt creates two datablocks (db0Guid and db1Guid). Then it sets the content of db0Guid to be an user-define value, while the content of db1Guid is set to be the guid value of db0Guid. Then the runtime 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 datablock db1Guid.

Once exampleEdtGuid starts executing function "exampleEdt", the contents of db1Guid are read. The function then retrieves the guid of datablock 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 datablock and finally calls "ocrShutdown".

B. OCR Change History

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