MIC COI API Reference Manual 0.65

Generated by Doxygen 1.6.1

Thu Jul 7 17:53:06 2011

Contents

1 MIC COI API Reference Manual 0.65

Disclaimer and Legal Information

Document Number:

World Wide Web: http://developer.intel.com

Intel Confidential

2 Disclaimer and Legal Information

Intel Confidential - This information contains highly sensitive technological or business information which could have a severely detrimental effect of disclosed to an unauthorized party.

All Intel Confidential media must be labeled and protected accordingly.

3 Coprocessor Offload Infrastructure Overview

3.1 Overview

The Coprocessor Offload Infrastructure (COI) for Knights Corner is a software library designed to ease the development of software tools and applications that run on a discrete Knights Corner device. The primary usage model is for applications that run on the host processor (e.g. Intel(R) Xeon(R) processor) to launch and communicate with workloads on one or more Knights Corner cards. But the COI model allows many different application configurations, including allowing applications running on Knights Corner to launch workloads on the host processor.

The COI model exposes a pipelined programming model to the user. This model allows workloads to be run and data to be moved asynchronously, allowing the host processor, device processor, and DMA engines to stay busy. In the COI pipelining model, work flows from a "source" to a "sink," either of which could be running on the host or device processors. Developers can configure one or more command pipelines to interact between sources and sinks. Commands on these pipelines are then run in an asynchronous, in-order fashion. This pipelined usage model exists in a number of offload environments, including graphics and network devices, and has been repeatedly shown to provide a balance between high performance and programmability. This model can also be used as the underpinnings on other popular programming models, including an RPC-like environment where device work is initiated by a large number of threads on the host.

The COI model is agnostic with respect to how applications and workloads are written. It is a C-language API that interacts with workloads through standard API entry points, but does not impose or provide a framework for exploiting vector or thread parallelism on the host or the device. This allows COI to be combined with any number of other programming models, including POSIX threads, Intel(R) Parallel Building Blocks, and Intel(R) compilers for both the host and the device.

3.2 Abstractions

COI exposes four key abstractions to users, allowing them to accomplish tasks that would be otherwise difficult to accomplish using just lower-level abstractions.

- COIEngine - This abstraction allows an application to enumerate the COI-capable devices in the system, including the host processor and any number of MIC devices in the same PCI root complex. The capabilities and dynamic load of the devices can be determined as well.
- COIProcess - The COIProcess abstraction encapsulates a process created by COI on a remote engine. Creating an instance of a COIProcess creates a user process on a remote engine, and having a process handle allows an application to create buffers and pipeline objects that can be used by the process.
- COIPipeline - A pipeline is a uni-directional, asynchronous command stream between COI processes. It allows remote functions to be run on a process running on another device. The process sending commands on a pipeline is called the "source" of the pipeline, and the process executing the commands is called the "sink" of the pipeline.
- COIBuffer - A COIBuffer object encapsulates data in a COI system. Buffers can be created with various properties that affect their behavior. For example, a buffers can be created such that its virtual address is the same no matter where it is used, allowing pointers to be used internally to the buffer. An application can use a COIBuffer without thinking about if the physical memory for the buffer is in device or host memory, or it can decide to exert control over placement and movement if the application's buffer usage model lends itself to a particular data movement scheme.

In addition to these key API abstractions, COI includes a few other useful abstractions. The COIBarrier abstraction allows for synchronization between asynchronous commands, including functions run on a COIPipeline. And the COIPerf and COISysInfo abstractions offer utility libraries for people programming MIC devices.

4 File and Function Naming Conventions

4.1 General Concepts

Files and APIs may contain multiple version numbers, and will always contain at least one. Occasionally, you will find a minor version on a file or API. This minor version number will increment with less disruptive changes to the contents of a file or an API: new functions, signature changes, special versions of a function, etc.

4.2 Header Files

There are three types of header files:

- Headers for APIs that are portable to both the source and sink. Such headers are named COI<description>[_minor_version]_common.h. COI_common.h and COIProcess_common.h are examples, and are found in <install_dir>/install/common.

- Headers for APIs that can only be used in sink-specific code (such as some COIBuffer APIs). Such headers are named COI<description>[_minor_version]_sink.h. Examples are COIBuffer_sink.h, and are found in <install_dir>/install/sink.
- Headers for APIs that only make sense on the source (such as COIEngine APIs for enumerating MIC devices). Such headers are named COI<description>[_minor_version]_source.h. Examples are COIEngine_source.h and COIBuffer_source.h, and are found in <install_dir>/install/source.

As noted, minor versions may be added to the filename when the contents change due to the addition of new functions.

4.3 APIs

APIs follow a similar naming scheme to header files. The first version of every function will have only a major version number. If significant functional changes or signature changes are made to a function, a new version of the function is created with a numbered suffix to indicate that it is similar to the original. Entirely new functions are given a new function name without a numbered suffix and the current major version number. Thus, you might see something like the following in a header file.

```
// *** Current Major Release 1:
COIFoo  (float a);

// The following are all variants of COIBar.
COIBar  (int x);
COIBar1 (int x, int y);
COIBar2 (int x, int y, int z);
```

By default, each function binds to the latest implementation with bug fixes and optimizations.

/*!

5 Module Documentation

5.1 COIBuffer

Modules

- COIBufferSource
- COIBufferSink

5.2 COIEngine

Modules

- COIEngineSource

5.2.1 Detailed Description

5.3 COIResult

Modules

- COIResultCommon

5.3.1 Detailed Description

5.4 COIPipeline

Modules

- COIPipelineSource
- COIPipelineSink

5.4.1 Detailed Description

5.5 COIProcess

Modules

- COIProcessSource
- COIProcessSink

5.5.1 Detailed Description

5.6 COIResultCommon

Enumerations

- enum COIRESULT {
  COI_SUCCESS = 0,
  COI_ERROR,
  COI_NOT_INITIALIZED,
  COI_ALREADY_INITIALIZED,
  COI_ALREADY_EXISTS,
  COI_DOES_NOT_EXIST,
  COI_INVALID_POINTER,
  COI_OUT_OF_RANGE,
  COI_NOT_SUPPORTED,
  COI_TIME_OUT_REACHED,
  COI_DUPLICATE_OBJECT,
  COI_ARGUMENT_MISMATCH,
  COI_SIZE_MISMATCH,
  COI_OUT_OF_MEMORY,
  COI_INVALID_HANDLE,
  COI_RETRY,
  COI_RESOURCE_EXHAUSTED,
  COI_ALREADY_LOCKED,
  COI_NOT_LOCKED,
  COI_MISSING_DEPENDENCY,

COI_UNDEFINED_SYMBOL,
COI_PENDING,
COI_BINARY_AND_HARDWARE_MISMATCH,
COI_PROCESS_DIED,
COI_INVALID_FILE,
COI_BARRIER_CANCELED,
COI_NUM_RESULTS }

Functions

- const char * COIResultGetName (COIRESULT in_ResultCode)
  Returns the string version of the passed in COIRESULT.

5.6.1 Enumeration Type Documentation

5.6.1.1 enum COIRESULT

Enumerator:

COI_SUCCESS The function succeeded without error.
COI_ERROR Unspecified error.
COI_NOT_INITIALIZED The function was called before the system was initialized.
COI_ALREADY_INITIALIZED The function was called after the system was initialized.
COI_ALREADY_EXISTS Cannot complete the request due to the existence of a similar object.
COI_DOES_NOT_EXIST The specified object was not found.
COI_INVALID_POINTER One of the provided addresses was not valid.
COI_OUT_OF_RANGE One of the arguments contains a value that is invalid.
COI_NOT_SUPPORTED This function is not currently supported as used.
COI_TIME_OUT_REACHED The specified time out caused the function to abort.
COI_DUPLICATE_OBJECT All objects must be unique.
COI_ARGUMENT_MISMATCH The specified arguments are not compatible.
COI_SIZE_MISMATCH The specified size does not match the expected size.
COI_OUT_OF_MEMORY The function was unable to allocate the required memory.
COI_INVALID_HANDLE One of the provided handles was not valid.
COI_RETRY This function currently can't complete, but might be able to later.
COI_RESOURCE_EXHAUSTED The resource was not large enough.
COI_ALREADY_LOCKED The object was expected to be unlocked, but was locked.
COI_NOT_LOCKED The object was expected to be locked, but was unlocked.
COI_MISSING_DEPENDENCY One or more dependent components could not be found.
COI_UNDEFINED_SYMBOL One or more symbols the component required was not defined in any library.

COI_PENDING Operation is not finished.

COI_BINARY_AND_HARDWARE_MISMATCH A specified binary will not run on the specified hardware.

COI_PROCESS_DIED

COI_INVALID_FILE The file is invalid for its intended usage in the function.

COI_BARRIER_CANCELED Barrier wait on a user barrier that was unregistered or is being unregistered returns COI_BARRIER_CANCELED.

COI_NUM_RESULTS Reserved, do not use.

Definition at line 20 of file COIResult_common.h.

5.6.2 Function Documentation

5.6.2.1 const char* COIResultGetName (COIRESULT *in_ResultCode*)

Returns the string version of the passed in COIRESULT. Thus if COI_RETRY is passed in, this function returns the string "COI_RETRY". If the error code passed ins is not valid then "COI_ERROR" will be returned.

Parameters:

in_ResultCode [in] COIRESULT code to return the string version of.

Returns:

String version of the passed in COIRESULT code.

5.7 COITypesSource

Data Structures

- struct coibarrier

Files

- file COITypes_common.h

Typedefs

- typedef uint32_t COI_CPU_MASK [8]
- typedef struct coibarrier COIBARRIER
- typedef struct coibuffer * COIBUFFER
- typedef struct coiengine * COIENGINE
- typedef struct coifunction * COIFUNCTION
- typedef struct coilibrary * COILIBRARY
- typedef struct coimapinst * COIMAPINSTANCE
- typedef struct coipipeline * COIPIPELINE
- typedef struct coiprocess * COIPROCESS

5.7.1 Typedef Documentation

5.7.1.1 typedef uint32_t COI_CPU_MASK[8]

Definition at line 37 of file COITypes_common.h.

5.7.1.2 typedef struct coibarrier COIBARRIER

Definition at line 32 of file COITypes_common.h.

5.7.1.3 typedef struct coibuffer* COIBUFFER

Definition at line 33 of file COITypes_common.h.

5.7.1.4 typedef struct coiengine* COIENGINE

Definition at line 31 of file COITypes_common.h.

5.7.1.5 typedef struct coifunction* COIFUNCTION

Definition at line 30 of file COITypes_common.h.

5.7.1.6 typedef struct coilibrary* COILIBRARY

Definition at line 34 of file COITypes_common.h.

5.7.1.7 typedef struct coimapinst* COIMAPINSTANCE

Definition at line 35 of file COITypes_common.h.

5.7.1.8 typedef struct coipipeline* COIPIPELINE

Definition at line 29 of file COITypes_common.h.

5.7.1.9 typedef struct coiprocess* COIPROCESS

Definition at line 28 of file COITypes_common.h.

5.8 COIPerfCommon

Files

- file COIPerf_common.h
  Performance Analysis API.

Functions

- __inline uint64_t COIPerfGetCycleCounter (void)
  Returns a performance counter value.
- uint64_t COIPerfGetCycleFrequency (void)
  Returns the calculated system frequency in hertz.

5.8.1 Function Documentation

5.8.1.1 __inline uint64_t COIPerfGetCycleCounter (void)

Returns a performance counter value. This function returns a performance counter value that increments at a constant rate for all time and is coherent across all cores.

Returns:

Current performance counter value or 0 if no performance counter is available

Definition at line 36 of file COIPerf_common.h.

5.8.1.2 uint64_t COIPerfGetCycleFrequency (void)

Returns the calculated system frequency in hertz.

Returns:

Current system frequency in hertz.

5.9 COISysInfoCommon

Files

- file COISysInfo_common.h

This interface allows developers to query the platform for system level information.

Defines

- #define INITIAL_APIC_ID_BITS 0xFF000000
- #define NUMBER_HW_THREADS 128

Functions

- uint32_t COISysGetAPICID (void)
- uint32_t COISysGetCoreCount (void)
- uint32_t COISysGetCoreIndex (void)
- uint32_t COISysGetHardwareThreadCount (void)
- uint32_t COISysGetHardwareThreadIndex (void)
- uint32_t COISysGetL2CacheCount (void)
- uint32_t COISysGetL2CacheIndex (void)

5.9.1 Define Documentation

5.9.1.1 #define INITIAL_APIC_ID_BITS 0xFF000000

Definition at line 28 of file COISysInfo_common.h.

5.9.1.2 #define NUMBER_HW_THREADS 128

Definition at line 29 of file COISysInfo_common.h.

5.9.2 Function Documentation

5.9.2.1 uint32_t COISysGetAPICID (void)

Returns:

The Advanced Programmable Interrupt Controller (APIC) ID of the hardware thread on which the caller is running.

Warning:

APIC IDs are unique to each hardware thread within a processor, but may not be sequential.

5.9.2.2 uint32_t COISysGetCoreCount (void)

Returns:

The number of cores exposed by the processor on which the caller is running.

5.9.2.3 uint32_t COISysGetCoreIndex (void)

Returns:

The index of the core on which the caller is running.

The indexes of neighboring cores will differ by a value of one and are within the range zero through COISysGetCoreCount()-1.

5.9.2.4 uint32_t COISysGetHardwareThreadCount (void)

Returns:

The number of hardware threads exposed by the processor on which the caller is running.

5.9.2.5 uint32_t COISysGetHardwareThreadIndex (void)

Returns:

The index of the hardware thread on which the caller is running.

The indexes of neighboring hardware threads will differ by a value of one and are within the range zero through COISysGetHardwareThreadCount()-1.

5.9.2.6 uint32_t COISysGetL2CacheCount (void)

Returns:

The number of level 2 caches within the processor on which the caller is running.

5.9.2.7 uint32_t COISysGetL2CacheIndex (void)

Returns:

The index of the level 2 cache on which the caller is running.

The indexes of neighboring cores will differ by a value of one and are within the range zero through COISysGetL2CacheCount()-1.

5.10 COIBarriercommon

Files

- file COIBarrier_common.h

Functions

- COIRESULT COIBarrierSignalUserBarrier (COIBARRIER in_Barrier)

  Signal one shot User barrier.

5.10.1 Function Documentation

5.10.1.1 COIRESULT COIBarrierSignalUserBarrier (COIBARRIER *in_Barrier*)

Signal one shot User barrier. User barriers created on source can be signaled from both sink and source. This fires the barrier and wakes up threads waiting on COIBarrierWait.

Note: For barriers that are not registered or already signaled this call will behave as a NOP. Users need to make sure that they pass valid barriers on the sink side.

Parameters:

in_Barrier Barrier Handle to be signaled.

Returns:

COI_SUCCESS

5.11 COIBarrierSource

Files

- file COIBarrier_source.h

Functions

- COIRESULT COIBarrierRegisterUserBarrier (COIBARRIER ∗out_pBarrier)

  Register a User COIBARRIER so that it can be fired.

- COIRESULT COIBarrierUnregisterUserBarrier (COIBARRIER in_Barrier)

  Unregister a User COIBARRIER.

- COIRESULT COIBarrierWait (uint16_t in_NumBarriers, const COIBARRIER ∗in_pBarriers, int32_t in_Timeout, uint8_t in_WaitForAll, uint32_t ∗out_pNumSignaled, uint32_t ∗out_pSignaledIndices)

  Wait for an arbitrary number of COIBARRIERs to be signaled as completed, eg when the run function or asynchronous map call associated with a barrier has finished execution.

5.11.1 Function Documentation

5.11.1.1 COIRESULT COIBarrierRegisterUserBarrier (COIBARRIER * *out_pBarrier*)

Register a User COIBARRIER so that it can be fired. Registered barrier is a one shot User barrier in other words once signaled it cannot be used again for signaling. You have to unregister and register again to enable signaling. A barrier will be reset if it is re-registered without unregistering, resulting in loss of all outstanding signals.

Parameters:

out_pBarrier Pointer to COIBARRIER handle being Registered

Returns:

COI_SUCCESS a barrier is successfully registered
COI_INVALID_POINTER if out_pBarrier is NULL

5.11.1.2 COIRESULT COIBarrierUnregisterUserBarrier (COIBARRIER *in_Barrier*)

Unregister a User COIBARRIER. Unregistering a unsignaled barrier is similar to firing a barrier. Except Calling COIBarrierWait on a barrier that is being unregistered returns COI_BARRIER_CANCELED

Parameters:

in_Barrier Barrier Handle to be unregistered.

Returns:

COI_SUCCESS a barrier is successfully registered

5.11.1.3 COIRESULT COIBarrierWait (uint16_t *in_NumBarriers*, const COIBARRIER * *in_pBarriers*, int32_t *in_Timeout*, uint8_t *in_WaitForAll*, uint32_t * *out_pNumSignaled*, uint32_t * *out_pSignaledIndices*)

Wait for an arbitrary number of COIBARRIERs to be signaled as completed, eg when the run function or asynchronous map call associated with a barrier has finished execution. If the user sets in_WaitForAll = True and not all of the barriers are signaled when the timeout period is reached then COI_TIME_OUT_-REACHED will be returned. If the user sets in_WaitForAll = False then if at least one barrier is signaled when the timeout is reached then COI_SUCCESS is returned.

Parameters:

in_NumBarriers [in] The number of barriers to wait for.

in_pBarriers [in] The array of COIBARRIER handles to wait for.

in_Timeout [in] The time in milliseconds to wait for the barrier. 0 polls and returns immediately, -1 blocks indefinitely.

in_WaitForAll [in] Boolean value specifying behavior. If true, wait for all barriers to be signaled, or for timeout, whichever happens first. If false, return when any barrier is signaled, or at timeout.

out_pNumSignaled [out] The number of barriers that were signaled. If in_NumBarriers is 1 or in_WaitForAll = True, this parameter is optional.

out_pSignaledIndices [out] Pointer to an array of indicies into the original barrier array. Those denoted have been signaled. The user must provide an array that is no smaller than the in_Barriers array. If in_NumBarriers is 1 or in_WaitForAll = True, this parameter is optional.

Returns:

COI_SUCCESS once a barrier has been signaled completed.
COI_TIME_OUT_REACHED if the barriers are still in use when the timeout is reached or timeout is zero (a poll).
COI_OUT_OF_RANGE if a negative value other than -1 is passed in to the in_Timeout parameter.
COI_OUT_OF_RANGE if in_NumBarriers is 0.
COI_INVALID_POINTER if in_pBarriers is NULL.
COI_ARGUMENT_MISMATCH if in_NumBarriers $>$ 1 and and if in_WaitForAll is not true and out_pSignaled or out_pSignaledIndicies are NULL .
COI_ARGUMENT_MISMATCH if out_pNumSignaled is not NULL and out_pSignaledIndices is NULL (or vice versa.)
COI_BARRIER_CANCELED if while waiting on a user barrier, it gets unregistered this returns COI_BARRIER_CANCELED

5.12 COIBufferSource

Enumerations

- enum COI_BUFFER_MOVE_FLAG {
  COI_BUFFER_MOVE_DATA = 0,
  COI_BUFFER_NO_MOVE_DATA }

  The buffer move flags are used to indicate when a buffer should be moved when it's state is changed.

- enum COI_BUFFER_STATE {
  COI_BUFFER_VALID = 0,
  COI_BUFFER_INVALID }

  The buffer states are used to indicate whether a buffer is available for access in a COIPROCESS.

- enum COI_BUFFER_TYPE {
  COI_BUFFER_NORMAL = 1,
  COI_BUFFER_STREAMING_TO_SINK,
  COI_BUFFER_STREAMING_TO_SOURCE,
  COI_BUFFER_PINNED }

  The valid buffer types that may be created using COIBufferCreate.

- enum COI_COPY_TYPE {
  COI_COPY_UNSPECIFIED = 0,
  COI_COPY_USE_DMA,
  COI_COPY_USE_CPU }

This matrix shows the valid combinations of buffer types and map operations that may be passed in to COIBufferMap.

- enum COI_MAP_TYPE {
  COI_MAP_READ_WRITE = 1,
  COI_MAP_READ_ONLY,
  COI_MAP_WRITE_ENTIRE_BUFFER }

  This matrix shows the valid combinations of buffer types and buffer flags that may be passed in to COIBufferCreate and COIBufferCreateFromMemory.

Functions

- COIRESULT COIBufferCopy (COIBUFFER in_DestBuffer, COIBUFFER in_SourceBuffer, uint64_t in_DestOffset, uint64_t in_SourceOffset, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data between two different buffers.

- COIRESULT COIBufferCreate (uint64_t in_Size, COI_BUFFER_TYPE in_Type, uint32_t in_Flags, const void *in_pInitData, uint32_t in_NumProcesses, const COIPROCESS *in_pProcesses, COIBUFFER *out_pBuffer)

  Creates a buffer that can be used in RunFunctions that are queued in pipelines.

- COIRESULT COIBufferCreateFromMemory (uint64_t in_Size, COI_BUFFER_TYPE in_Type, uint32_t in_Flags, void *in_Memory, uint32_t in_NumProcesses, const COIPROCESS *in_pProcesses, COIBUFFER *out_pBuffer)

  Creates a buffer from some existing memory that can be used in RunFunctions that are queued in pipelines.

- COIRESULT COIBufferDestroy (COIBUFFER in_Buffer)

  Destroys a buffer.

- COIRESULT COIBufferGetSinkAddress (COIBUFFER in_Buffer, uint64_t *out_pAddress)

  Gets the Sink's virtual address of the buffer.

- COIRESULT COIBufferMap (COIBUFFER in_Buffer, uint64_t in_Offset, uint64_t in_Length, COI_MAP_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion, COIMAPINSTANCE *out_pMapInstance, void **out_ppData)

  This call initiates a request to access a region of a buffer.

- COIRESULT COIBufferRead (COIBUFFER in_SourceBuffer, uint64_t in_Offset, void *in_pDestData, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data from a buffer into local memory.

- COIRESULT COIBufferSetState (COIBUFFER in_Buffer, COIPROCESS in_Process, COI_BUFFER_STATE in_State, COI_BUFFER_MOVE_FLAG in_DataMove)

  This API allows an experienced COI developer to set where a COIBUFFER is located and when the COIBUFFER's data is moved.

- COIRESULT COIBufferUnmap (COIMAPINSTANCE in_MapInstance, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Disables Source access to the region of the buffer that was provided through the corresponding call to COIBufferMap.

- COIRESULT COIBufferWrite (COIBUFFER in_DestBuffer, uint64_t in_Offset, const void *in_pSourceData, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data from a normal virtual address into an existing COIBUFFER.

COIBUFFER creation flags.

Please see the COI_VALID_BUFFER_TYPES_AND_FLAGS matrix below which describes the valid combinations of buffer types and flags.

- #define COI_OPTIMIZE_SINK_READ 0x00000010

  Hint to the runtime that the sink will frequently read the buffer.

- #define COI_OPTIMIZE_SINK_WRITE 0x00000020

  Hint to the runtime that the sink will frequently write the buffer.

- #define COI_OPTIMIZE_SOURCE_READ 0x00000004

  Hint to the runtime that the source will frequently read the buffer.

- #define COI_OPTIMIZE_SOURCE_WRITE 0x00000008

  Hint to the runtime that the source will frequently write the buffer.

- #define COI_SAME_ADDRESS_SINKS 0x00000001

  Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated.

- #define COI_SAME_ADDRESS_SINKS_AND_SOURCE 0x00000002

  Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated and in the source process.

- #define COI_SPU_TXS_MEMORY 0x00000040

  Causes the buffer to be allocated from memory that is accessible by the SPU and TXS devices on MIC.

5.12.1 Define Documentation

5.12.1.1 #define COI_OPTIMIZE_SINK_READ 0x00000010

Hint to the runtime that the sink will frequently read the buffer.

Definition at line 77 of file COIBuffer_source.h.

5.12.1.2 #define COI_OPTIMIZE_SINK_WRITE 0x00000020

Hint to the runtime that the sink will frequently write the buffer.

Definition at line 80 of file COIBuffer_source.h.

5.12.1.3 #define COI_OPTIMIZE_SOURCE_READ 0x00000004

Hint to the runtime that the source will frequently read the buffer.

Definition at line 71 of file COIBuffer_source.h.

5.12.1.4 #define COI_OPTIMIZE_SOURCE_WRITE 0x00000008

Hint to the runtime that the source will frequently write the buffer.

Definition at line 74 of file COIBuffer_source.h.

5.12.1.5 #define COI_SAME_ADDRESS_SINKS 0x00000001

Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated.

Definition at line 64 of file COIBuffer_source.h.

5.12.1.6 #define COI_SAME_ADDRESS_SINKS_AND_SOURCE 0x00000002

Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated and in the source process.

Definition at line 68 of file COIBuffer_source.h.

5.12.1.7 #define COI_SPU_TXS_MEMORY 0x00000040

Causes the buffer to be allocated from memory that is accessible by the SPU and TXS devices on MIC. This flag is only valid for buffers used in run functions that execute on the MIC device. Note that SPU/TXS buffers are not currently supported

Definition at line 86 of file COIBuffer_source.h.

5.12.2 Enumeration Type Documentation

5.12.2.1 enum COI_BUFFER_MOVE_FLAG

The buffer move flags are used to indicate when a buffer should be moved when it's state is changed. This is used with COIBufferSetState.

Enumerator:

COI_BUFFER_MOVE_DATA

COI_BUFFER_NO_MOVE_DATA

Definition at line 209 of file COIBuffer_source.h.

5.12.2.2 enum COI_BUFFER_STATE

The buffer states are used to indicate whether a buffer is available for access in a COIPROCESS. This is used with COIBufferSetState.

Enumerator:

COI_BUFFER_VALID

COI_BUFFER_INVALID

Definition at line 201 of file COIBuffer_source.h.

5.12.2.3 enum COI_BUFFER_TYPE

The valid buffer types that may be created using COIBufferCreate. Please see the COI_VALID_BUFFER_-TYPES_AND_FLAGS matrix below which describes the valid combinations of buffer types and flags.

Enumerator:

COI_BUFFER_NORMAL Normal buffers exist as a single physical buffer in either Source or Sink physical memory. Mapping the buffer may stall the pipelines.

COI_BUFFER_STREAMING_TO_SINK A streaming buffer creates new versions each time it is passed to Runfunction. These new versions are consumed by run functions. To_SINK buffers are used to send data from SOURCE to SINK These buffers are SOURCE write only buffers. If read, won't get Data written by SINK

COI_BUFFER_STREAMING_TO_SOURCE To_SOURCE buffers are used to get data from SINK to SOURCE These buffers are SOURCE Read only buffers. If written, data won't get reflected on SINK side.

COI_BUFFER_PINNED A pinned buffer exists in a shared memory region and is always available for read or write operations.

Definition at line 31 of file COIBuffer_source.h.

5.12.2.4 enum COI_COPY_TYPE

This matrix shows the valid combinations of buffer types and map operations that may be passed in to COIBufferMap.

```
static const uint64_t
COI_VALID_BUFFER_TYPES_AND_MAP
[COI_BUFFER_PINNED+1][COI_MAP_WRITE_ENTIRE_BUFFER+1] = {
/*                       | MAP   | MAP   | MAP
                         | READ  | READ  | WRITE
                         | WRITE | ONLY  | ENTIRE|
                         +-------+-------+-------*/
MMM(INVALID              ,   F   ,   F   ,   F   ),
MMM(NORMAL               ,   T   ,   T   ,   T   ),
MMM(STREAMING_TO_SINK    ,   F   ,   F   ,   T   ),
MMM(STREAMING_TO_SOURCE ,   F   ,   T   ,   F   ),

};
```

The valid copy operation types for the COIBufferWrite, COIBufferRead, and COIBufferCopy APIs.

Enumerator:

COI_COPY_UNSPECIFIED The runtime can pick the best suitable way to copy the data.

COI_COPY_USE_DMA The runtime should use DMA to copy the data.

COI_COPY_USE_CPU The runtime should use a CPU copy to copy the data.

Definition at line 184 of file COIBuffer_source.h.

5.12.2.5 enum COI_MAP_TYPE

This matrix shows the valid combinations of buffer types and buffer flags that may be passed in to COIBufferCreate and COIBufferCreateFromMemory.

```
static const uint64_t
COI_VALID_BUFFER_TYPES_AND_FLAGS[COI_BUFFER_PINNED+1] = {
/*                               | SAME  |
                         | SAME  | ADDR  | OPT   | OPT   | OPT   | OPT   | SPU  |
                         | ADDR  | SINK  | SRC   | SRC   | SINK  | SINK  | TXS  |
                         | SINKS | SRC   | READ  | WRITE | READ  | WRITE | MEM  |
                         +-------+-------+-------+-------+-------+-------+------*/
MTM(INVALID              ,   F   ,   F   ,   F   ,   F   ,   F   ,   F   ,   F  ),
MTM(NORMAL               ,   T   ,   T   ,   T   ,   T   ,   T   ,   T   ,   T  ),
MTM(STREAMING_TO_SINK    ,   T   ,   F   ,   F   ,   T   ,   T   ,   T   ,   F  ),
MTM(STREAMING_TO_SOURCE,   T   ,   F   ,   T   ,   T   ,   F   ,   T   ,   F  ),

};
```

These flags control how the buffer will be accessed on the source after it is mapped. Please see the COI_-VALID_BUFFER_TYPES_AND_MAP matrix below for the valid buffer type and map operation combinations.

Enumerator:

COI_MAP_READ_WRITE Allows the application to read and write the contents of the buffer after it is mapped.

COI_MAP_READ_ONLY If this flag is set then the application must only read from the buffer after it is mapped. If the application writes to the buffer the contents will not be reflected back to the sink or stored for the next time the buffer is mapped on the source. This allows the runtime to make significant performance optimizations in buffer handling.

COI_MAP_WRITE_ENTIRE_BUFFER Setting this flag means that the source will overwrite the entire buffer once it is mapped. The app must not read from the buffer and must not expect the contents of the buffer to be synchronized from the sink side during the map operation. This allows the runtime to make significant performance optimizations in buffer handling.

Definition at line 128 of file COIBuffer_source.h.

5.12.3 Function Documentation

5.12.3.1 COIRESULT COIBufferCopy (COIBUFFER *in_DestBuffer*, COIBUFFER *in_SourceBuffer*, uint64_t *in_DestOffset*, uint64_t *in_SourceOffset*, uint64_t *in_Length*, COI_COPY_TYPE *in_Type*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, COIBARRIER * *out_pCompletion*)

Copy data between two different buffers.

Parameters:

in_DestBuffer [in] Buffer to copy into.

in_SourceBuffer [in] Buffer to copy from.

in_DestOffset [in] Location in the destination buffer to start writing to. Currently must be a page aligned value. For now, all buffers start on a page

in_SourceOffset [in] Location in the source buffer to start reading from. Currently must be a page aligned value. For now, all buffers start on a page

in_Length [in] The number of bytes to copy from in_SourceBuffer into in_DestinationBuffer. If the length is specified as zero then the entire buffer will be copied. Must not be larger than the size of in_SourceBuffer or in_DestBuffer and must not over run in_SourceBuffer or in_DestBuffer if offsets are specified. Currently must be a page aligned value.

in_Type [in] The type of copy operation to use, one of either COI_COPY_UNSPECIFIED, COI_-COPY_USE_DMA, COI_COPY_USE_CPU.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the copy call to wait for any additional barriers to be signaled before starting the copy operation.

in_pDependencies [in] An optional array of handles to previously created COIBARRIER objects that this copy operation will wait for before starting. This allows the user to create dependencies between buffer copy calls and other operations such as run functions and map calls. The user may pass in NULL if they do not wish to wait for any additional dependencies to complete before doing the copy.

out_pCompletion [out] An optional barrier to be signaled when the copy has completed. This barrier can be used as a dependency to order the copy with regard to future operations. If no completion barrier is passed in then the copy is synchronous and will block until the transfer is complete.

Returns:

COI_SUCCESS if the buffer was copied successfully.
COI_INVALID_HANDLE if either buffer handle was invalid.

COI_DUPLICATE_OBJECT if in_SourceBuffer and in_DestBuffer refer to the same object.
COI_OUT_OF_RANGE if in_DestOffset is is beyond the end of in_DestBuffer
COI_OUT_OF_RANGE if in_SourceOffset is beyond the end of in_SourceBuffer.
COI_OUT_OF_RANGE if in_DestOffset + in_Length exceeds the size of the in_DestBuffer
COI_OUT_OF_RANGE if in_SourceOffset + in_Length exceeds the size of in_SourceBuffer.
COI_OUT_OF_RANGE in_DestOffset, in_SourceOffset or in_Length are not page aligned.
COI_ARGUMENT_MISMATCH if the in_pDependencies is non NULL but in_NumDependencies is 0.
COI_ARGUMENT_MISMATCH if in_pDependencies is NULL but in_NumDependencies is not 0.
COI_NOT_SUPPORTED if either buffer is of type COI_BUFFER_STREAMING_TO_SINK or COI_BUFFER_STREAMING_TO_SOURCE.
COI_RETRY if in_DestBuffer or in_SourceBuffer are mapped and not COI_BUFFER_PINNED buffers.

5.12.3.2 COIRESULT COIBufferCreate (uint64_t *in_Size*, COI_BUFFER_TYPE *in_Type*, uint32_t *in_Flags*, const void * *in_pInitData*, uint32_t *in_NumProcesses*, const COIPROCESS * *in_pProcesses*, COIBUFFER * *out_pBuffer*)

Creates a buffer that can be used in RunFunctions that are queued in pipelines. The address space for the buffer is reserved when it is created although the memory may not be committed until the buffer is used for the first time.

Parameters:

in_Size [in] The number of bytes to allocate for the buffer. If in_Size is not page aligned, it will be rounded up.

in_Type [in] The type of the buffer to create.

in_Flags [in] A bitmask of attributes for the newly created buffer. Some of these flags are required for correctness while others are provided as hints to the runtime system so it can make certain performance optimizations.

in_pInitData [in] If non-NULL the buffer will be initialized with the data pointed to by pInitData. The memory at in_pInitData must hold at least in_Size bytes.

in_NumProcesses [in] The number of processes with which this buffer might be used.

in_pProcesses [in] An array of COIPROCESS handles identifying the processes with which this buffer might be used.

out_pBuffer [out] Pointer to a buffer handle. The handle will be filled in with a value that uniquely identifies the newly created buffer. This handle should be disposed of via COIBufferDestroy() once it is no longer needed.

Returns:

COI_SUCCESS if the buffer was created
COI_ARGUMENT_MISMATCH if the in_Type and in_Flags parameters are not compatible with one another. Please see the COI_VALID_BUFFER_TYPES_AND_FLAGS map above for information about which flags and types are compatible.
COI_OUT_OF_RANGE if in_Size is zero, if the bits set in the in_Flags parameter are not recognized flags, or if in_NumProcesses is zero.
COI_INVALID_POINTER if the in_pProcesses or out_pBuffer parameter is NULL.
COI_INVALID_HANDLE if one of the COIPROCESS handles in the in_pProcesses array does not identify a valid process.

COI_OUT_OF_MEMORY if allocating the buffer fails.
COI_RESOURCE_EXHAUSTED if the device is out of buffer memory.

5.12.3.3 COIRESULT COIBufferCreateFromMemory (uint64_t *in_Size*, COI_BUFFER_TYPE *in_Type*, uint32_t *in_Flags*, void * *in_Memory*, uint32_t *in_NumProcesses*, const COIPROCESS * *in_pProcesses*, COIBUFFER * *out_pBuffer*)

Creates a buffer from some existing memory that can be used in RunFunctions that are queued in pipelines. The memory provided is used as backing store for the buffer on the source and must not be freed before the buffer is destroyed. The runtime system may also reserve memory on the sink for the buffer to optimize performance. While the user still owns the memory passed in they must use COIBufferMap calls to get access to the memory so that the runtime knows when the memory has been modified. If the user just writes directly to the memory location then those changes may not be visible on the sink when the corresponding buffer is accessed. Whatever values are already present in the memory location when this call is made is preserved. The memory values are also preserved when COIBufferDestroy is called.

Parameters:

in_Size [in] The size of in_Memory in bytes. If in_Size is not page aligned, it will be rounded up.

in_Type [in] The type of the buffer to create. Note that streaming buffers can not be created from user memory. Only COI_BUFFER_NORMAL and COI_BUFFER_PINNED buffer types are supported.

in_Flags [in] A bitmask of attributes for the newly created buffer. Some of these flags are required for correctness while others are provided as hints to the runtime system so it can make certain performance optimizations. Note that the flag COI_SAME_ADDRESS_SINKS_AND_SOURCE is still valid but may fail if the same address as in_Memory can not be allocated on the sink.

in_Memory [in] A pointer to an already allocated memory region on the source that should be turned into a COIBUFFER. Although the user still owns this memory they should not free it before calling COIBufferDestroy. They must also only access the memory using COIBUFFER semantics, for example using COIBufferMap/COIBufferUnmap when they wish to read or write the data. There are no alignment or size requirements for this memory region.

in_NumProcesses [in] The number of processes with which this buffer might be used.

in_pProcesses [in] An array of COIPROCESS handles identifying the processes with which this buffer might be used.

out_pBuffer [out] Pointer to a buffer handle. The handle will be filled in with a value that uniquely identifies the newly created buffer. This handle should be disposed of via COIBufferDestroy() once it is no longer needed.

Returns:

COI_SUCCESS if the buffer was created
COI_NOT_SUPPORTED if the in_Type value is not COI_BUFFER_NORMAL or COI_BUFFER_-PINNED.
COI_ARGUMENT_MISMATCH if the in_Type and in_Flags parameters are not compatible with one another. Please see the COI_VALID_BUFFER_TYPES_AND_FLAGS map above for information about which flags and types are compatible.
COI_OUT_OF_RANGE if in_Size is zero, if the bits set in the in_Flags parameter are not recognized flags, or if in_NumProcesses is zero.
COI_INVALID_POINTER if in_Memory, in_pProcesses or out_pBuffer parameter is NULL.
COI_INVALID_HANDLE if one of the COIPROCESS handles in the in_pProcesses array does not identify a valid process.

5.12.3.4 COIRESULT COIBufferDestroy (COIBUFFER *in_Buffer*)

Destroys a buffer. Will block on completion of any operations on the buffer, such as COIPipelineRunFunction or COIBufferCopy. Will block until all COIBufferAddRef calls have had a matching COIBufferReleaseRef call made. Will not block on an outstanding COIBufferUnmap but will instead return COI_RETRY.

Parameters:

in_Buffer [in] Handle of the buffer to destroy.

Returns:

COI_SUCCESS if the buffer was destroyed.
COI_INVALID_HANDLE if the buffer handle was invalid.
COI_RETRY if the buffer is currently mapped. The buffer must first be unmapped before it can be destroyed.

5.12.3.5 COIRESULT COIBufferGetSinkAddress (COIBUFFER *in_Buffer*, uint64_t * *out_pAddress*)

Gets the Sink's virtual address of the buffer. This is the same address that is passed to the run function on the Sink. This address is only valid on the Sink and should not be dereferenced on the Source (except for the special case of buffers created with the COI_SAME_ADDRESS flag).

Parameters:

in_Buffer [in] Buffer handle

out_pAddress [out] pointer to a uint64_t* that will be filled with the address.

Returns:

COI_SUCCESS upon successful return of the buffer's address.
COI_INVALID_HANDLE if the passed in buffer handle was invalid.
COI_INVALID_POINTER if the out_pAddress parameter was invalid.

5.12.3.6 COIRESULT COIBufferMap (COIBUFFER *in_Buffer*, uint64_t *in_Offset*, uint64_t *in_Length*, COI_MAP_TYPE *in_Type*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, COIBARRIER * *out_pCompletion*, COIMAPINSTANCE * *out_pMapInstance*, void ** *out_ppData*)

This call initiates a request to access a region of a buffer. Multiple overlapping (or non overlapping) regions can be mapped simultaneously for any given buffer. If a completion barrier is specified this call will queue a request for the data which will be satisfied when the buffer is available. Once all conditions are met the completion barrier will be signaled and the user can access the data at out_ppData. The user can call COIBarrierWait with out_pCompletion to find out when the map operation has completed. If the user accesses the data before the map operation is complete the results are undefined. If out_pCompletion is NULL then this call blocks until the map operation completes and when this call returns out_ppData can

be safely accessed. This call returns a map instance handle in an out parameter which must be passed into COIBufferUnmap when the user no longer needs access to that region of the buffer.

Note that different types of buffers behave differently when mapped. For instance, mapping a COI_-BUFFER_NORMAL for write must stall if the buffer is currently being written to by a run function. Mapping a COI_BUFFER_STREAMING_TO_SINK will create a new physical copy of the buffer and make it available immediately. Mapping a COI_BUFFER_PINNED buffer will not affect other functions that use that buffer since a COI_BUFFER_PINNED buffer can be mapped at any time. The asynchronous operation of COIBufferMap will likely be most useful when paired with a COI_BUFFER_NORMAL.

Parameters:

in_Buffer [in] Handle for the buffer to map.

in_Offset [in] Offset into the buffer that a pointer should be returned for. The value 0 can be passed in to signify that the mapped region should start at the beginning of the buffer.

in_Length [in] Length of the buffer area to map. This parameter, in combination with in_Offset, allows the caller to specify that only a subset of an entire buffer need be mapped. A value of 0 can be passed in to signify that the mapped region should reach the end of the buffer. Thus, for a 4096 byte buffer, if in_Offset is 2048, and in_Length is 0, then the bytes in positions 2049-4095 will be mapped.

in_Type [in] The access type that is needed by the application. This will affect how the data can be accessed once the map operation completes. See the COI_MAP_TYPE enum for more details.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the map call initiation to wait for any barriers to be signaled before starting the map operations.

in_pDependencies [in] An optional array of handles to previously created COIBARRIER objects that this map operation will wait for before starting. This allows the user to create dependencies between asynchronous map calls and other operations such as run functions or other asynchronous map calls. The user may pass in NULL if they do not wish to wait for any dependencies to complete before initiating map operations.

out_pCompletion [out] An optional pointer to a COIBARRIER object that will be signaled when a map call with the passed in buffer would complete immediately, that is, the buffer memory has been allocated on the host and its contents updated. The user may pass in NULL if the user wants COIBufferMap to perform a blocking map operation.

out_pMapInstance [out] A pointer to a COIMAPINSTANCE which represents this mapping of the buffer and must be passed in to COIBufferUnmap when access to this region of the buffer data is no longer needed.

out_ppData [out] Pointer to the buffer data. The data will only be valid when the completion object is signaled, or for a synchronous map operation with the call to map returns.

Returns:

COI_SUCCESS if the map request succeeds.
COI_OUT_OF_RANGE if in_Offset is beyond the end of the buffer.
COI_OUT_OF_RANGE if in_Offset + in_Length exceeds the size of the buffer.
COI_ARGUMENT_MISMATCH if in_NumDependencies is non-zero while in_pDependencies was passed in as NULL.
COI_ARGUMENT_MISMATCH if in_pDependencies is non-NULL but in_NumDependencies is zero.
COI_INVALID_HANDLE if in_Buffer is not a valid buffer handle.
COI_INVALID_POINTER if out_pMapInstance or out_ppData is NULL.

5.12.3.7 COIRESULT COIBufferRead (COIBUFFER *in_SourceBuffer*, uint64_t *in_Offset*, void * *in_pDestData*, uint64_t *in_Length*, COI_COPY_TYPE *in_Type*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, COIBARRIER * *out_pCompletion*)

Copy data from a buffer into local memory.

Parameters:

in_SourceBuffer [in] Buffer to write into.

in_Offset [in] Location in the buffer to start reading from. Currently must be a page aligned value. For now, all buffers start on a page boundary.

in_pDestData [in] A pointer to local memory that should be written into from the provided buffer.

in_Length [in] The number of bytes to write from in_SourceBuffer into in_pDestData. Must not be larger than the size of in_SourceBuffer and must not over run in_SourceBuffer if an in_Offset is provided. Currently must be a page aligned value.

in_Type [in] The type of copy operation to use, one of either COI_COPY_UNSPECIFIED, COI_COPY_USE_DMA, COI_COPY_USE_CPU.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the read call to wait for any additional barriers to be signaled before starting the read operation.

in_pDependencies [in] An optional array of handles to previously created COIBARRIER objects that this read operation will wait for before starting. This allows the user to create dependencies between buffer read calls and other operations such as run functions and map calls. The user may pass in NULL if they do not wish to wait for any additional dependencies to complete before doing the read.

out_pCompletion [out] An optional barrier to be signaled when the copy has completed. This barrier can be used as a dependency to order the copy with regard to future operations. If no completion barrier is passed in then the copy is synchronous and will block until the transfer is complete.

Returns:

COI_SUCCESS if the buffer was copied successfully.
COI_INVALID_HANDLE if the buffer handle was invalid.
COI_OUT_OF_RANGE if in_Offset is beyond the end of the buffer.
COI_ARGUMENT_MISMATCH if the in_pDependencies is non NULL but in_NumDependencies is 0.
COI_ARGUMENT_MISMATCH if in_pDependencies is NULL but in_NumDependencies is not 0.
COI_OUT_OF_RANGE if in_Offset + in_Length exceeds the size of the buffer.
COI_OUT_OF_RANGE in_Offset or in_Length are not page aligned.
COI_OUT_OF_RANGE if in_Length is 0.
COI_INVALID_POINTER if the in_pDestData pointer is NULL.
COI_RETRY if in_SourceBuffer is mapped and is not a COI_BUFFER_PINNED buffer.

5.12.3.8 COIRESULT COIBufferSetState (COIBUFFER *in_Buffer*, COIPROCESS *in_Process*, COI_BUFFER_STATE *in_State*, COI_BUFFER_MOVE_FLAG *in_DataMove*)

This API allows an experienced COI developer to set where a COIBUFFER is located and when the COIBUFFER's data is moved. This functionality is useful when the developer knows when and where

a buffer is going to be accessed. It allows the data movement to happen sooner than if the COI runtime tried to manage the buffer placement itself. The advantage of this API is that the developer knows much more about their own application's data access patterns and can therefore optimize the data access to be much more efficient than the COI runtime. Using this API may yield better memory utilization, lower latency and overall improved workload throughput. This API does respect implicit dependencies for buffer read/write hazards. For example, if the buffer is being written in one COIPROCESS and the user requests the buffer be placed in another COIPROCESS then this API will wait for the first access to complete before moving the buffer. This API is not required for program correctness. It is intended solely for advanced COI developers who wish to fine tune their application performance.

Parameters:

in_Buffer [in] The buffer to modify.

in_Process [in] The process where the state is being modified for this buffer.

in_State [in] The new state for the buffer. The buffer's state could be set to invalid on one of the sink processes where it is being used.

in_DataMove [in] A flag to indicate if the buffer's data should be moved when the state is changed. For instance, a buffer's state may be set to valid on a process and the data move flag may be set to COI_BUFFER_MOVE_DATA which would cause the buffer contents to be copied to the process where it is now valid.

Returns:

COI_SUCCESS if the buffer's state was changed successfully.
COI_INVALID_HANDLE if in_Buffer or in_Process is invalid.

5.12.3.9 COIRESULT COIBufferUnmap (COIMAPINSTANCE *in_MapInstance*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, COIBARRIER * *out_pCompletion*)

Disables Source access to the region of the buffer that was provided through the corresponding call to COIBufferMap. The number of calls to COIBufferUnmap() should always match the number of calls made to COIBufferMap(). The data pointer returned from the COIBufferMap() call will be invalid after this call.

Parameters:

in_MapInstance [in] buffer map instance handle to unmap.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the unmap call to wait for any barriers to be signaled before performing the unmap operation.

in_pDependencies [in] An optional array of handles to previously created COIBARRIER objects that this unmap operation will wait for before starting. This allows the user to create dependencies between asynchronous unmap calls and other operations such as run functions or other asynchronous unmap calls. The user may pass in NULL if they do not wish to wait for any dependencies to complete before initiating unmap operations.

out_pCompletion [out] An optional pointer to a COIBARRIER object that will be signaled when the unmap is complete. The user may pass in NULL if the user wants COIBufferUnmap to perform a blocking unmap operation.

Returns:

COI_SUCCESS upon successful unmapping of the buffer instance.
COI_INVALID_HANDLE if the passed in map instance handle was NULL.
COI_ARGUMENT_MISMATCH if the in_pDependencies is non NULL but in_NumDependencies is 0.
COI_ARGUMENT_MISMATCH if in_pDependencies is NULL but in_NumDependencies is not 0.

5.12.3.10 COIRESULT COIBufferWrite (COIBUFFER *in_DestBuffer*, uint64_t *in_Offset*, const void * *in_pSourceData*, uint64_t *in_Length*, COI_COPY_TYPE *in_Type*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, COIBARRIER * *out_pCompletion*)

Copy data from a normal virtual address into an existing COIBUFFER.

Parameters:

in_DestBuffer [in] Buffer to write into.

in_Offset [in] Location in the buffer to start writing to. Currently must be a page aligned value. For now, all buffers start on a page boundary.

in_pSourceData [in] A pointer to local memory that should be copied into the provided buffer.

in_Length [in] The number of bytes to write from in_pSourceData into in_DestBuffer. Must not be larger than the size of in_DestBuffer and must not over run in_DestBuffer if an in_Offset is provided. Currently must be a page aligned value.

in_Type [in] The type of copy operation to use, one of either COI_COPY_UNSPECIFIED, COI_COPY_USE_DMA, COI_COPY_USE_CPU.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the write call to wait for any additional barriers to be signaled before starting the write operation.

in_pDependencies [in] An optional array of handles to previously created COIBARRIER objects that this write operation will wait for before starting. This allows the user to create dependencies between buffer write calls and other operations such as run functions and map calls. The user may pass in NULL if they do not wish to wait for any additional dependencies to complete before doing the write.

out_pCompletion [out] An optional barrier to be signaled when the copy has completed. This barrier can be used as a dependency to order the copy with regard to future operations. If no completion barrier is passed in then the copy is synchronous and will block until the transfer is complete.

Returns:

COI_SUCCESS if the buffer was copied successfully.
COI_INVALID_HANDLE if the buffer handle was invalid.
COI_OUT_OF_RANGE if in_Offset is beyond the end of the buffer.
COI_ARGUMENT_MISMATCH if the in_pDependencies is non NULL but in_NumDependencies is 0.
COI_ARGUMENT_MISMATCH if in_pDependencies is NULL but in_NumDependencies is not 0.
COI_OUT_OF_RANGE in_Offset or in_Length are not page aligned.
COI_INVALID_POINTER if the in_pSourceData pointer is NULL.
COI_OUT_OF_RANGE if in_Offset + in_Length exceeds the size of the buffer.
COI_OUT_OF_RANGE if in_Length is 0.
COI_RETRY if in_DestBuffer is mapped and is not a COI_BUFFER_PINNED buffer.

5.13 COIEngineSource

Data Structures

- struct COI_ENGINE_INFO

  This structure returns information about a MIC engine.

Defines

- #define COI_MAX_DRIVER_VERSION_STR_LEN 255
- #define COI_MAX_HW_THREADS 1024
- #define COI_MAX_ISA_KNC_DEVICES 32
- #define COI_MAX_ISA_KNF_DEVICES 32
- #define COI_MAX_ISA_x86_64_DEVICES 1

Enumerations

- enum COI_ISA_TYPE {
  COI_ISA_INVALID = 0,
  COI_ISA_x86_64,
  COI_ISA_KNF,
  COI_ISA_KNC }

Functions

- COIRESULT COIEngineGetCount (COI_ISA_TYPE in_ISA, uint32_t *out_pNumEngines)

  Returns the number of engines in the system that match the provided ISA.

- COIRESULT COIEngineGetHandle (COI_ISA_TYPE in_ISA, uint32_t in_EngineIndex, COIENGINE *out_pEngineHandle)

  Returns the handle of a user specified engine.

- COIRESULT COIEngineGetInfo (COIENGINE in_EngineHandle, COI_ENGINE_INFO *out_pEngineInfo)

  Returns information related to a specified engine.

5.13.1 Define Documentation

5.13.1.1 #define COI_MAX_DRIVER_VERSION_STR_LEN 255

Definition at line 27 of file COIEngine_source.h.

5.13.1.2 #define COI_MAX_HW_THREADS 1024

Definition at line 29 of file COIEngine_source.h.

5.13.1.3 #define COI_MAX_ISA_KNC_DEVICES 32

Definition at line 33 of file COIEngine_source.h.

5.13.1.4 #define COI_MAX_ISA_KNF_DEVICES 32

Definition at line 32 of file COIEngine_source.h.

5.13.1.5 #define COI_MAX_ISA_x86_64_DEVICES 1

Definition at line 31 of file COIEngine_source.h.

5.13.2 Enumeration Type Documentation

5.13.2.1 enum COI_ISA_TYPE

Enumerator:

COI_ISA_INVALID
COI_ISA_x86_64
COI_ISA_KNF
COI_ISA_KNC

Definition at line 35 of file COIEngine_source.h.

5.13.3 Function Documentation

5.13.3.1 COIRESULT COIEngineGetCount (COI_ISA_TYPE *in_ISA*, uint32_t * *out_pNumEngines*)

Returns the number of engines in the system that match the provided ISA.

Parameters:

in_ISA [in] The bitmask specifying the ISA of the engines the caller would like to enumerate. Only the number of engines that match a subset of the specified bitmask will be returned to the user.

out_pNumEngines [out] The number of engines available. This can be used to index into the engines using COIEngineGetHandle().

Returns:

COI_SUCCESS if the function completed without error.
COI_DOES_NOT_EXIST if the in_ISA parameter is not valid.
COI_INVALID_POINTER if the out_pNumEngines parameter is NULL.

5.13.3.2 COIRESULT COIEngineGetHandle (COI_ISA_TYPE *in_ISA*, uint32_t *in_EngineIndex*, COIENGINE * *out_pEngineHandle*)

Returns the handle of a user specified engine.

Parameters:

in_ISA [in] The bitmask specifying the ISA of the engine. Only an engine that matches a subset of the specified bitmask will be returned to the user.

in_EngineIndex A unsigned integer which specifies the zero-based position of the engine in a collection of engines. The makeup of this collection is defined by the in_ISA parameter.

out_pEngineHandle The address of an COIENGINE handle.

Returns:

COI_SUCCESS if the function completed without error.
COI_DOES_NOT_EXIST if the in_ISA parameter is not valid.
COI_OUT_OF_RANGE if in_EngineIndex is greater than or equal to the number of engines that match the in_ISA parameter.
COI_INVALID_POINTER if the out_pEngineHandle parameter is NULL.

5.13.3.3 COIRESULT COIEngineGetInfo (COIENGINE *in_EngineHandle*, COI_ENGINE_INFO * *out_pEngineInfo*)

Returns information related to a specified engine.

Parameters:

in_EngineHandle [in] The COIENGINE structure as provided from COIEngineGetHandle() which to query for device level information.

out_pEngineInfo [out] The address of a user allocated COI_ENGINE_INFO structure. Upon success, the contents of the structure will be updated to contain information related to the specified engine.

Returns:

COI_SUCCESS if the function completed without error.
COI_INVALID_HANDLE if the in_EngineHandle handle is not valid.
COI_INVALID_POINTER if the out_pEngineInfo pointer is NULL.

5.14 COIPipelineSource

Files

- file COIPipeline_source.h

Defines

- #define COI_PIPELINE_MAX_IN_BUFFERS 32768
- #define COI_PIPELINE_MAX_IN_MISC_DATA_LEN 32768

Enumerations

- enum COI_ACCESS_FLAGS {
  COI_SINK_READ = 1,
  COI_SINK_WRITE,
  COI_SINK_WRITE_ENTIRE }
  These flags specify how a buffer will be used within a run function.

Functions

- COIRESULT COIPipelineClearCPUMask (COI_CPU_MASK *in_Mask)
  Clears a given mask.
- COIRESULT COIPipelineCreate (COIPROCESS in_Process, COI_CPU_MASK in_Mask, uint32_t in_StackSize, COIPIPELINE *out_pPipeline)
  Create a pipeline assoiated with a remote process.
- COIRESULT COIPipelineDestroy (COIPIPELINE in_Pipeline)
  Destroys the inidicated pipeline, releasing its resources.
- COIRESULT COIPipelineFlush (COIPIPELINE in_Pipeline, int32_t in_Timeout)
  Flushes the commands in the pipeline, causing the Sink to start processing run functions from the indicated pipeline, and waits for all run functions that have been queued on the pipeline before the flush to finish before the call returns.
- COIRESULT COIPipelineGetEngine (COIPIPELINE in_Pipeline, COIENGINE *out_pEngine)
  Retrieve the engine that the pipeline is associated with.
- COIRESULT COIPipelineRunFunction (COIPIPELINE in_Pipeline, COIFUNCTION in_Function, uint32_t in_NumBuffers, const COIBUFFER *in_Buffers, const COI_ACCESS_FLAGS *in_pBufferAccessFlags, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, const void *in_pMiscData, uint16_t in_MiscDataLen, void *out_pAsyncReturnValue, uint16_t in_AsyncReturnValueLen, COIBARRIER *out_pCompletion)
  Enqueues a function in the remote process binary to be executed.
- COIRESULT COIPipelineSetCPUMask (COIPROCESS in_Process, uint32_t in_CoreID, uint8_t in_ThreadID, COI_CPU_MASK *out_pMask)
  Set a given mask to a particular core:thread pair.

5.14.1 Define Documentation

5.14.1.1 #define COI_PIPELINE_MAX_IN_BUFFERS 32768

Definition at line 48 of file COIPipeline_source.h.

5.14.1.2 #define COI_PIPELINE_MAX_IN_MISC_DATA_LEN 32768

Definition at line 49 of file COIPipeline_source.h.

5.14.2 Enumeration Type Documentation

5.14.2.1 enum COI_ACCESS_FLAGS

These flags specify how a buffer will be used within a run function. They allow COI to make optimizations in how it moves data around the system. These flags can affect the correctness of an application, so they must be set properly. For example, if a buffer is used in a run function with the COI_SINK_READ flag and then mapped on the source, COI may use a previously cached version of the buffer instead of retrieving data from the sink.

Enumerator:

COI_SINK_READ Specifies that the run function will only read the associated buffer.

COI_SINK_WRITE Specifies that the run function will only write the associated buffer.

COI_SINK_WRITE_ENTIRE Specifies that the run function will overwrite the entire associated buffer and therefore the buffer will not be synchronized with the source before execution.

Definition at line 34 of file COIPipeline_source.h.

5.14.3 Function Documentation

5.14.3.1 COIRESULT COIPipelineClearCPUMask (COI_CPU_MASK * *in_Mask*)

Clears a given mask.

Parameters:

in_Mask [in] Pointer to the mask to clear.

Returns:

COI_SUCCESS if the mask was cleared.
COI_INVALID_POINTER if in_Mask is invalid.

5.14.3.2 COIRESULT COIPipelineCreate (COIPROCESS *in_Process*, COI_CPU_MASK *in_Mask*, uint32_t *in_StackSize*, COIPIPELINE * *out_pPipeline*)

Create a pipeline assoiated with a remote process. This pipeline can then be used to execute remote functions and to share data using COIBuffers.

Parameters:

in_Process [in] A handle to an already existing process that the pipeline will be associated with.

in_Mask [in] An optional mask of the set of hardware threads on which the sink pipeline command processing thread could run.

in_StackSize [in] An optional value that will be used when the pipeline processing thread is created on the sink. If the user passes in 0 the OS default stack size will be used.

out_pPipeline [out] Handle returned to uniquely identify the pipeline that was created for use in later API calls.

Returns:

COI_SUCCESS if the pipeline was successfully created.
COI_INVALID_HANDLE if the in_Process handle passed in was invalid.
COI_INVALID_POINTER if the out_pPipeline pointer was NULL.
COI_RESOURCE_EXHAUSTED if no more COIPipelines can be created.
COI_TIME_OUT_REACHED if establishing the communication channel with the remote pipeline timed out.

5.14.3.3 COIRESULT COIPipelineDestroy (COIPIPELINE *in_Pipeline*)

Destroys the inidicated pipeline, releasing its resources.

Parameters:

in_Pipeline [in] Pipeline to destroy.

Returns:

COI_SUCCESS if the pipeline was destroyed
COI_INVALID_HANDLE if the in_Pipeline handle passed in was invalid.

5.14.3.4 COIRESULT COIPipelineFlush (COIPIPELINE *in_Pipeline*, int32_t *in_Timeout*)

Flushes the commands in the pipeline, causing the Sink to start processing run functions from the indicated pipeline, and waits for all run functions that have been queued on the pipeline before the flush to finish before the call returns.

Parameters:

in_Pipeline [in] Pipeline to flush.

in_Timeout [in] The time in milliseconds to wait for the flush to complete. -1 waits indefinitely, 0 does not wait at all and returns immediately.

Returns:

COI_SUCCESS if the pipeline was flushed.
COI_INVALID_HANDLE if the pipeline handle passed in was invalid.
COI_OUT_OF_RANGE if in_Timeout is less than -1.
COI_TIME_OUT_REACHED if the flush has not completed when the timeout is reached.

5.14.3.5 COIRESULT COIPipelineGetEngine (COIPIPELINE *in_Pipeline*, COIENGINE * *out_pEngine*)

Retrieve the engine that the pipeline is associated with.

Parameters:

in_Pipeline [in] Pipeline to query.

out_pEngine [out] The handle of the Engine.

Returns:

COI_SUCCESS if the engine was retrieved.
COI_INVALID_HANDLE if the pipeline handle passed in was invalid.
COI_INVALID_POINTER if the out_pEngine parameter is NULL.

5.14.3.6 COIRESULT COIPipelineRunFunction (COIPIPELINE *in_Pipeline*, COIFUNCTION *in_Function*, uint32_t *in_NumBuffers*, const COIBUFFER * *in_Buffers*, const COI_ACCESS_FLAGS * *in_pBufferAccessFlags*, uint32_t *in_NumDependencies*, const COIBARRIER * *in_pDependencies*, const void * *in_pMiscData*, uint16_t *in_MiscDataLen*, void * *out_pAsyncReturnValue*, uint16_t *in_AsyncReturnValueLen*, COIBARRIER * *out_pCompletion*)

Enqueues a function in the remote process binary to be executed. The function execution is asynchronous in regards to the Source and all run functions enqueued on a pipeline are executed in-order. The run function will only execute when all of the required buffers are present in the Sink's memory.

Parameters:

in_Pipeline [in] Handle to a previously created pipeline that this run function should be enqueued to.

in_Function [in] Previously returned handle from a call to COIPipelineGetFunctionHandle() that represents a function in the application running on the Sink process.

in_NumBuffers [in] The number of buffers that are being passed to the run function. This number must match the number of buffers in the in_Buffers and in_pBufferAccessFlags arrays. Must be less than COI_PIPELINE_MAX_IN_BUFFERS.

in_Buffers [in] An array of COIBUFFER handles that the function is expected to use during its execution. Each buffer when it arrives at the Sink process will be at least 4k page aligned, thus, using a very large number of small buffers is memory inefficient and should be avoided.

in_pBufferAccessFlags [in] An array of flag values which correspond to the buffers passed in the in_Buffers parameter. These flags are used to track dependendencies between different run functions being executed from different pipelines.

in_NumDependencies [in] The number of dependencies specified in the in_pDependencies array. This may be 0 if the caller does not want the run function to wait for any dependencies.

in_pDependencies [in] An optional array of COIBARRIER objects that this run function will wait for before executing. This allows the user to create dependencies between run functions in different pipelines. The user may pass in NULL if they do not wish to wait for any dependencies to complete.

in_pMiscData [in] Pointer to user defined data, typically used to pass parameters to Sink side functions. Should only be used for small amounts data since the data will be placed directly in the Driver's command buffer. COIBuffers should be used to pass large amounts of data.

in_MiscDataLen [in] Size of the in_pMiscData in bytes. Must be less than COI_PIPELINE_MAX_IN_MISC_DATA_LEN, and should usually be much smaller, see documentation for the parameter in_pMiscData.

out_pAsyncReturnValue [out] Pointer to user-allocated memory where the return value from the run function will be placed. This memory should not be read until out_pCompletion has been signalled.

in_AsyncReturnValueLen [in] Size of the out_pAsyncReturnValue in bytes.

out_pCompletion [out] An optional pointer to a COIBARRIER object that will be signaled when this run function has completed execution. The user may pass in NULL if they do not wish to signal any COIBARRIERs when this run function completes.

Returns:

COI_SUCCESS if the function was successfully placed in a pipeline for future execution. Note that the actual execution of the function will occur in the future.
COI_OUT_OF_RANGE if in_NumBuffers is greater than COI_PIPELINE_MAX_IN_BUFFERS or if in_MiscDataLen is greater than COI_PIPELINE_MAX_IN_MISC_DATA_LEN.
COI_INVALID_HANDLE if the pipeline handle passed in was invalid.
COI_INVALID_HANDLE if the function handle passed in was invalid.
COI_INVALID_HANDLE if any of the buffers passed in are invalid.
COI_ARGUMENT_MISMATCH if in_NumDependencies is non-zero while in_pDependencies was passed in as NULL.
COI_ARGUMENT_MISMATCH if in_pDependencies is non-NULL but in_NumDependencies is zero.
COI_ARGUMENT_MISMATCH if in_MiscDataLen is non-zero while in_pMiscData was passed in as NULL.
COI_ARGUMENT_MISMATCH if in_pMiscData is non-NULL but in_MiscDataLen is zero.
COI_ARGUMENT_MISMATCH if in_NumBuffers is non-zero and in_Buffers or in_pBufferAccessFlags are NULL.
COI_ARGUMENT_MISMATCH if in_pBufferAccessFlags is non-NULL but in_NumBuffers is zero.
COI_ARGUMENT_MISMATCH if in_ReturnValueLen is non-zero while in_pReturnValue was passed in as NULL.
COI_ARGUMENT_MISMATCH if in_pReturnValue is non-NULL but in_ReturnValueLen is zero.
COI_RETRY if any input buffers, which are not pinned buffers, are still mapped when passed to the run function.

5.14.3.7 COIRESULT COIPipelineSetCPUMask (COIPROCESS *in_Process*, uint32_t *in_CoreID*, uint8_t *in_ThreadID*, COI_CPU_MASK * *out_pMask*)

Set a given mask to a particular core:thread pair.

Parameters:

in_Process [in] A handle to an already existing process that the pipeline will be associated with.

in_CoreID [in] Core to affinitize to; must be less than the number of cores on the device.

in_ThreadID [in] Thread on the core to affinitize to (0 - 3).

out_pMask [out] Pointer to the mask to set.

Returns:

COI_SUCCESS if the mask was set.
COI_OUT_OF_RANGE if the in_CoreID or in_ThreadID is out of range.
COI_INVALID_POINTER if out_pMask is invalid.
COI_INVALID_HANDLE if in_Process is invalid.

5.15 COIProcessSource

Files

- file COIProcess_source.h

Defines

- #define COI_MAX_FILE_NAME_LENGTH 256
- #define COI_MAX_FUNCTION_NAME_LENGTH 256
- #define COI_PROCESS_SOURCE ((COIPROCESS)-1)

  This is a special COIPROCESS handle that can be used to indicate that the source process should be used for an operation.

Enumerations

- enum COI_SHUTDOWN_REASON {
  COI_SHUTDOWN_OK = 0,
  COI_SHUTDOWN_SIGTERM,
  COI_SHUTDOWN_SEGFAULT }

Functions

- COIRESULT COIProcessCreateFromFile (COIENGINE in_Engine, const char *in_pBinaryName, int in_Argc, const char **in_ppArgv, uint8_t in_DupEnv, const char **in_ppAdditionalEnv, uint8_t in_ProxyActive, const char *in_ProxyRoot, uint64_t in_BufferSpace, COIPROCESS *out_pProcess)

  Create a remote process on the Sink and start executing it's main() function.

- COIRESULT COIProcessCreateFromMemory (COIENGINE in_Engine, const char *in_pBinaryName, const void *in_pBinaryBuffer, uint64_t in_BinaryBufferLength, int in_Argc, const char **in_ppArgv, uint8_t in_DupEnv, const char **in_ppAdditionalEnv, uint8_t in_ProxyActive, const char *in_ProxyRoot, uint64_t in_BufferSpace, COIPROCESS *out_pProcess)

  Create a remote process on the Sink and start executing it's main() function.

- COIRESULT COIProcessDestroy (COIPROCESS in_Process, int32_t in_WaitForMainTimeout, uint8_t in_ForceDestroy, int8_t *out_pProcessReturn, COI_SHUTDOWN_REASON *out_pReason)

  Destroys the indicated process, releasing its resources.

- COIRESULT COIProcessGetFunctionHandles (COIPROCESS in_Process, uint32_t in_NumFunctions, const char **in_ppFunctionNameArray, COIFUNCTION *out_pFunctionHandleArray)

Given a loaded native process, gets an array of function handles that can be used to schedule run functions on a pipeline associated with that process.

- COIRESULT COIProcessLoadLibraryFromFile (COIPROCESS in_Process, const char *in_pFileName, const char *in_pLibraryName, COILIBRARY *out_pLibrary)

  Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows.

- COIRESULT COIProcessLoadLibraryFromMemory (COIPROCESS in_Process, const void *in_pLibraryBuffer, uint64_t in_LibraryBufferLength, const char *in_pLibraryName, COILIBRARY *out_pLibrary)

  Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows.

- COIRESULT COIProcessUnLoadLibrary (COIPROCESS in_Process, COILIBRARY in_Library)

  Unloads a a previously loaded shared library from the specified remote process.

5.15.1 Define Documentation

5.15.1.1 #define COI_MAX_FILE_NAME_LENGTH 256

Definition at line 40 of file COIProcess_source.h.

5.15.1.2 #define COI_MAX_FUNCTION_NAME_LENGTH 256

Definition at line 354 of file COIProcess_source.h.

5.15.1.3 #define COI_PROCESS_SOURCE ((COIPROCESS)-1)

This is a special COIPROCESS handle that can be used to indicate that the source process should be used for an operation.

Definition at line 31 of file COIProcess_source.h.

5.15.2 Enumeration Type Documentation

5.15.2.1 enum COI_SHUTDOWN_REASON

Enumerator:

COI_SHUTDOWN_OK
COI_SHUTDOWN_SIGTERM
COI_SHUTDOWN_SEGFAULT

Definition at line 33 of file COIProcess_source.h.

5.15.3 Function Documentation

5.15.3.1 COIRESULT COIProcessCreateFromFile (COIENGINE *in_Engine*, const char * *in_pBinaryName*, int *in_Argc*, const char ** *in_ppArgv*, uint8_t *in_DupEnv*, const char ** *in_ppAdditionalEnv*, uint8_t *in_ProxyActive*, const char * *in_ProxyRoot*, uint64_t *in_BufferSpace*, COIPROCESS * *out_pProcess*)

Create a remote process on the Sink and start executing it's main() function. This will also automatically load any dependent shared objects on to the device. Once the process is created, remote calls can be initiated by using the RunFunction mechanism found in the COIPipeline APIs. For more information on how dependencies are loaded, see the COIProcessLoadLibrary APIs.

Parameters:

in_Engine [in] A handle retrieved via a call to COIEngineGetHandle() that indicates which device to create the process on. This is necessary because there can be more than one device within the system.

in_pBinaryName [in] Pointer to a null-terminated string that contains the path to the program binary to be instantiated as a process on the sink device. The file name will be accessed via fopen and fread, as such, the passed in binary name must be locatable via these commands. Also, the file name (without directory information) will be used automatically by the system to create the argv[0] of the new process.

in_Argc [in] The number of arguments being passed in to the process in the in_ppArgv parameter.

in_ppArgv [in] An array of strings that represent the arguments being passed in. The system will auto-generate argv[0] using in_pBinaryName and thus that parameter cannot be passed in using in_ppArgv. Instead, in_ppArgv contains the rest of the parameters being passed in.

in_DupEnv [in] A boolean that indicates whether the process that is being created should inherit the environment of the caller.

in_ppAdditionalEnv [in] An array of strings that represent additional environment variables. This parameter must terminate the array with a NULL string. For convenience it is also allowed to be NULL if there are no additional environment variables that need adding. Note that any environment variables specified here will be in addition to but override those that were inherited via in_DupEnv.

in_ProxyActive [in] A boolean that specifies whether the process that is to be created wants I/O proxy support.

in_ProxyRoot [in] If proxy support was requested this string indicates the root directory that will be prepended to any proxy file I/O. If proxy support was requested passing NULL will set the proxy root to the default value of "/".

in_BufferSpace [in] The most memory (in bytes) that will ever be allocated for buffers that will be used by pipelines associated with this process. If the buffer space specified is 0, then all buffers will be allocated in process space upon allocation, without limit. If it is specified then: Buffer memory is allocated by the sink process as part of the process creation. Buffer creations that specify a buffer size larger than the buffer space of one of its processes will fail. Run functions that specify a COIBuffer collection larger than this limit will fail. Run functions whose buffer space, when combined with AddRef'd buffers and buffers from other pipelines, exceed the buffer space limit will stall until enough buffer space is released.

out_pProcess [out] Handle returned to uniquely identify the process that was created for use in later API calls.

Returns:

COI_SUCCESS if the remote process was successfully created.
COI_INVALID_HANDLE if the in_Engine handle passed in was invalid.
COI_INVALID_POINTER if out_pProcess was NULL.
COI_INVALID_POINTER if in_pBinaryName was NULL.
COI_DOES_NOT_EXIST if in_pBinaryName cannot be found.
COI_BINARY_AND_HARDWARE_MISMATCH if in_pBinaryName is an invalid executable on the engine specified.
COI_RESOURCE_EXHAUSTED if no more COIProcesses can be created.
COI_ARGUMENT_MISMATCH if in_Argc is 0 and in_ppArgv is not NULL.
COI_ARGUMENT_MISMATCH if in_Argc is greater than 0 and in_ppArgv is NULL.
COI_OUT_OF_RANGE if in_Argc is less than 0.
COI_OUT_OF_RANGE if the length of in_pBinaryName is greater than or equal to COI_MAX_-FILE_NAME_LENGTH.
COI_ARGUMENT_MISMATCH if in_ProxyActive is false and in_ProxyRoot is not NULL.
COI_DOES_NOT_EXIST if in_ProxyRoot is not NULL and does not exist.
COI_TIME_OUT_REACHED if establishing the communication channel with the remote process timed out.

5.15.3.2 COIRESULT COIProcessCreateFromMemory (COIENGINE *in_Engine*, const char * *in_pBinaryName*, const void * *in_pBinaryBuffer*, uint64_t *in_BinaryBufferLength*, int *in_Argc*, const char ** *in_ppArgv*, uint8_t *in_DupEnv*, const char ** *in_ppAdditionalEnv*, uint8_t *in_ProxyActive*, const char * *in_ProxyRoot*, uint64_t *in_BufferSpace*, COIPROCESS * *out_pProcess*)

Create a remote process on the Sink and start executing it's main() function. This will also automatically load any dependent shared objects on to the device. Once the process is created, remote calls can be initiated by using the RunFunction mechanism found in the COIPipeline APIs. For more information on how dependencies are loaded, see the COIProcessLoadLibrary APIs.

Parameters:

in_Engine [in] A handle retrieved via a call to COIEngineGetHandle() that indicates which device to create the process on. This is necessary because there can be more than one device within the system.

in_pBinaryName [in] Pointer to a null-terminated string that contains the name to give the process that will be created. Note that the final name will strip out any directory information from in_-pBinaryName and use the file information to generate an argv[0] for the new process.

in_pBinaryBuffer [in] Pointer to a buffer whose contents represent the sink-side process that we want to create.

in_BinaryBufferLength [in] Number of bytes in in_pBinaryBuffer.

in_Argc [in] The number of arguments being passed in to the process in the in_ppArgv parameter.

in_ppArgv [in] An array of strings that represent the arguments being passed in. The system will auto-generate argv[0] using in_pBinaryName and thus that parameter cannot be passed in using in_ppArgv. Instead, in_ppArgv contains the rest of the parameters being passed in.

in_DupEnv [in] A boolean that indicates whether the process that is being created should inherit the environment of the caller.

in_ppAdditionalEnv [in] An array of strings that represent additional environment variables. This parameter must terminate the array with a NULL string. For convenience it is also allowed

to be NULL if there are no additional environment variables that need adding. Note that any environment variables specified here will be in addition to but override those that were inherited via in_DupEnv.

in_ProxyActive [in] A boolean that specifies whether the process that is to be created wants I/O proxy support.

in_ProxyRoot [in] If proxy support was requested this string indicates the root directory that will be prepended to any proxy file I/O. If proxy support was requested passing NULL will set the proxy root to the default value of "/".

in_BufferSpace [in] The most memory (in bytes) that will ever be allocated for buffers that will be used by pipelines associated with this process. If the buffer space specified is 0, then all buffers will be allocated in process space upon allocation, without limit. If it is specified then: Buffer memory is allocated by the sink process as part of the process creation. Buffer creations that specify a buffer size larger than the buffer space of one of its processes will fail. Run functions that specify a COIBuffer collection larger than this limit will fail. Run functions whose buffer space, when combined with AddRef'd buffers and buffers from other pipelines, exceed the buffer space limit will stall until enough buffer space is released.

out_pProcess [out] Handle returned to uniquely identify the process that was created for use in later API calls.

Returns:

COI_SUCCESS if the remote process was successfully created.
COI_INVALID_HANDLE if the in_Engine handle passed in was invalid.
COI_INVALID_POINTER if out_pProcess was NULL.
COI_INVALID_POINTER if in_pBinaryName or in_pBinaryBuffer was NULL.
COI_BINARY_AND_HARDWARE_MISMATCH if in_pBinaryName is an invalid executable on the engine specified.
COI_RESOURCE_EXHAUSTED if no more COIProcesses can be created.
COI_ARGUMENT_MISMATCH if in_Argc is 0 and in_ppArgv is not NULL.
COI_ARGUMENT_MISMATCH if in_Argc is greater than 0 and in_ppArgv is NULL.
COI_OUT_OF_RANGE if in_Argc is less than 0.
COI_OUT_OF_RANGE if the length of in_pBinaryName is greater than or equal to COI_MAX_-FILE_NAME_LENGTH.
COI_OUT_OF_RANGE if in_BinaryBufferLength is 0.
COI_ARGUMENT_MISMATCH if in_ProxyActive is false and in_ProxyRoot is not NULL.
COI_DOES_NOT_EXIST if in_ProxyRoot is not NULL and does not exist.
COI_TIME_OUT_REACHED if establishing the communication channel with the remote process timed out.

5.15.3.3 COIRESULT COIProcessDestroy (COIPROCESS *in_Process*, int32_t *in_WaitForMainTimeout*, uint8_t *in_ForceDestroy*, int8_t * *out_pProcessReturn*, COI_SHUTDOWN_REASON * *out_pReason*)

Destroys the indicated process, releasing its resources.

Parameters:

in_Process [in] Process to destroy.

in_WaitForMainTimeout [in] The number of milliseconds to wait for the main() function to return in the sink process before timing out. If 0 is passed in this function polls and immediately returns.

If -1 is passed in this function waits indefinitely for main() to return before freeing the process resources.

in_ForceDestroy [in] If this flag is set to true then the sink process will be forcibly terminated after the timeout has been reached. A timeout value of 0 will kill the process immediately, while a timeout of -1 is invalid. If the flag is set to false then a message will be sent to the sink process requesting a clean shutdown. In most cases this flag should be set to false. If a sink process is not responding then it may be necessary to set this flag to true.

out_pProcessReturn [out] The value returned from the main() function executing in the sink process. This is an optional parameter. If the caller is not interested in the return value from the remote process they may pass in NULL for this parameter.

out_pReason [out] This parameter specifies the shutdown reason. This may be COI_SHUTDOWN_OK if the remote process exited cleanly or some other value if the process exited abnormally. This is an optional parameter and the caller may pass in NULL if they are not interested in the shutdown reason.

Returns:

COI_SUCCESS if the process was destroyed.
COI_INVALID_HANDLE if the process handle passed in was invalid.
COI_OUT_OF_RANGE for any negative in_WaitForMainTimeout value except -1.
COI_ARGUMENT_MISMATCH if in_WaitForMainTimeout is -1 and in_ForceDestroy is true.
COI_TIME_OUT_REACHED if the sink process is still running after waiting in_WaitForMainTimeout milliseconds and in_ForceDestroy is false. This is true even if in_WaitForMainTimeout was 0. In this case, out_pProcessReturn and out_pReason are undefined.

5.15.3.4 COIRESULT COIProcessGetFunctionHandles (COIPROCESS *in_Process*, uint32_t *in_NumFunctions*, const char ** *in_ppFunctionNameArray*, COIFUNCTION * *out_pFunctionHandleArray*)

Given a loaded native process, gets an array of function handles that can be used to schedule run functions on a pipeline associated with that process. See the documentation for COIPipelineRunFunction() for additional information. All functions that are to be retrieved in this fashion must have the define COINATIVEPROCESSEXPORT preceeding their type specification. For functions that are written in C++, either the entries in in_pFunctionNameArray in must be pre-mangled, or the functions must be declared as extern "C". It is possible for this call to successfully find function handles for some of the names passed in but not all of them. If this occurs COI_DOES_NOT_EXIST will return and any handles not found will be returned as NULL.

Parameters:

in_Process [in] Process handle previously returned via COIProcessCreate()

in_NumFunctions [in] Number of function names passed in to the in_pFunctionNames array.

in_ppFunctionNameArray [in] Pointer to an array of null-terminated strings that match the name of functions present in the code of the binary previously loaded via COIProcessCreate(). Note that if a C++ function is used, then the string passed in must already be properly name-mangled, or extern "C" must be used for where the function is declared.

out_pFunctionHandleArray [in out] Pointer to a location created by the caller large enough to hold an array of COIFUNCTION sized elements that has in_numFunctions entries in the array.

Returns:

COI_SUCCESS if all function names indicated were found.
COI_INVALID_HANDLE if the in_Process handle passed in was invalid.
COI_OUT_OF_RANGE if in_NumFunctions is zero.
COI_INVALID_POINTER if the in_ppFunctionNameArray or out_pFunctionHandleArray pointers was NULL.
COI_DOES_NOT_EXIST if one or more function names were not found. To determine the function names that were not found, check which elements in the out_pFunctionHandleArray are set to NULL.
COI_OUT_OF_RANGE if any of the null-terminated strings passed in via in_ppFunctionNameArray were more than COI_MAX_FUNCTION_NAME_LENGTH characters in length including the null.

Warning:

This operation can take several milliseconds so it is recommended that it only be be done at load time.

5.15.3.5 COIRESULT COIProcessLoadLibraryFromFile (COIPROCESS *in_Process*, const char * *in_pFileName*, const char * *in_pLibraryName*, COILIBRARY * *out_pLibrary*)

Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows. Dependencies for this library that are not listed with absolute paths are searched for first in current working directory, then in the colon-delimited paths in the environment variable SINK_LD_LIBRARY_PATH, and finally in the MPSS-installed library paths.

Parameters:

in_Process [in] Process to load the library into.

in_pFileName [in] The name of the shared library file on the source's file system that is being loaded. If the file name is not an absolute path, the file is searched for in the same manner as dependencies.

in_pLibraryName [in] Name for the shared library. This optional parameter can be specified in case the dynamic library doesn't have an SO_NAME field. If specified, it will take precedence over the SO_NAME if it exists. If it is not specified then the library must have a valid SO_NAME field.

out_pLibrary [out] If COI_SUCCESS or COI_ALREADY_EXISTS is returned, the handle that uniquely identifies the loaded library.

Returns:

COI_SUCCESS if the library was successfully loaded.
COI_INVALID_HANDLE if the process handle passed in was invalid.
COI_INVALID_POINTER if in_pFileName is NULL.
COI_DOES_NOT_EXIST if in_pFileName cannot be found.
COI_INVALID_FILE if the file is not a valid shared library.
COI_ARGUMENT_MISMATCH if the shared library is missing an SONAME and in_pLibraryName is NULL.
COI_ARGUMENT_MISMATCH if in_pLibraryName is the same as that of any of the dependencies (recursive) of the library being loaded.
COI_ALREADY_EXISTS if there is an existing COILIBRARY handle that identifies this library, and this COILIBRARY hasn't been unloaded yet.
COI_BINARY_AND_HARDWARE_MISMATCH if the binary's target machine does not match the engine associated with in_Process.
COI_INVALID_POINTER if out_pLibrary is NULL.

5.15.3.6 COIRESULT COIProcessLoadLibraryFromMemory (COIPROCESS *in_Process*, const void * *in_pLibraryBuffer*, uint64_t *in_LibraryBufferLength*, const char * *in_pLibraryName*, COILIBRARY * *out_pLibrary*)

Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows. Dependencies for this library that are not listed with absolute paths are searched for first in current working directory, then in the colon-delimited paths in the environment variable SINK_LD_LIBRARY_PATH, and finally in the MPSS-installed library paths.

Parameters:

in_Process [in] Process to load the library into.

in_pLibraryBuffer [in] The memory buffer containing the shared library to load.

in_LibraryBufferLength [in] The number of bytes in the memory buffer in_pLibraryBuffer.

in_pLibraryName [in] Name for the shared library. This optional parameter can be specified in case the dynamic library doesn't have an SO_NAME field. If specified, it will take precedence over the SO_NAME if it exists. If it is not specified then the library must have a valid SO_NAME field.

out_pLibrary [out] If COI_SUCCESS or COI_ALREADY_EXISTS is returned, the handle that uniquely identifies the loaded library.

Returns:

COI_SUCCESS if the library was successfully loaded.
COI_INVALID_HANDLE if the process handle passed in was invalid.
COI_OUT_OF_RANGE if in_LibraryBufferLength is 0.
COI_INVALID_FILE if in_pLibraryBuffer does not represent a valid shared library file.
COI_ARGUMENT_MISMATCH if the shared library is missing an SONAME and in_pLibraryName is NULL.
COI_ARGUMENT_MISMATCH if in_pLibraryName is the same as that of any of the dependencies (recursive) of the library being loaded.
COI_ALREADY_EXISTS if there is an existing COILIBRARY handle that identifies this library, and this COILIBRARY hasn't been unloaded yet.
COI_BINARY_AND_HARDWARE_MISMATCH if the binary's target machine does not match the engine associated with in_Process.
COI_INVALID_POINTER if out_pLibrary is NULL.

5.15.3.7 COIRESULT COIProcessUnLoadLibrary (COIPROCESS *in_Process*, COILIBRARY *in_Library*)

Unloads a a previously loaded shared library from the specified remote process.

Parameters:

in_Process [in] Process that we are unloading a library from.

in_Library [in] Library that we want to unload.

Returns:

COI_SUCCESS if the library was successfully loaded.
COI_INVALID_HANDLE if the process or library handle were invalid.

5.16 COIBufferSink

Functions

- COIRESULT COIBufferAddRef (void *in_pBuffer)

  Adds a reference to the memory of a buffer.

- COIRESULT COIBufferReleaseRef (void *in_pBuffer)

  Removes a reference to the memory of a buffer.

5.16.1 Function Documentation

5.16.1.1 COIRESULT COIBufferAddRef (void * *in_pBuffer*)

Adds a reference to the memory of a buffer. The memory of the buffer will remain on the device until both a corresponding COIBufferReleaseRef() call is made and the run function that delivered the buffer returns.

Warning:

It is possible for enqueued run functions to be unable to execute due to all card memory being occupied by addref'ed buffers. As such, it is important that whenever a buffer is addref'd that there be no dependencies on future run functions for progress to be made towards releasing the buffer.

Parameters:

in_pBuffer [in] Pointer to the start of a buffer being addref'ed, that was passed in at the start of the run function.

Returns:

COI_SUCCESS if the buffer ref count was successfully incremented.
COI_INVALID_POINTER if the buffer pointer was invalid.

5.16.1.2 COIRESULT COIBufferReleaseRef (void * *in_pBuffer*)

Removes a reference to the memory of a buffer. The memory of the buffer will be eligible for being freed on the device when the following conditions are met: the run function that delivered the buffer returns, and the number of calls to COIBufferReleaseRef() matches the number of calls to COIBufferAddRef().

Parameters:

in_pBuffer [in] Pointer to the start of a buffer previously addref'ed, that was passed in at the start of the run function.

Returns:

COI_SUCCESS if the buffer refcount was successfully decremented.
COI_INVALID_POINTER if the buffer pointer was invalid.
COI_OUT_OF_RANGE if the buffer did not have COIBufferAddRef() previously called on it.

5.17 COIPipelineSink

Files

- file COIPipeline_sink.h

Typedefs

- typedef void(∗ RunFunctionPtr_t)(uint32_t in_BufferCount, void ∗∗in_ppBufferPointers, uint64_t ∗in_pBufferLengths, void ∗in_pMiscData, uint16_t in_MiscDataLength, void ∗in_pReturnValue, uint16_t in_ReturnValueLength)

  This is the prototype that run functions should follow.

Functions

- COIRESULT COIPipelineStartExecutingRunFunctions ()

  Start processing pipelines on the Sink.

5.17.1 Typedef Documentation

5.17.1.1 typedef void(∗ RunFunctionPtr_t)(uint32_t in_BufferCount, void ∗∗in_ppBufferPointers, uint64_t ∗in_pBufferLengths, void ∗in_pMiscData, uint16_t in_MiscDataLength, void ∗in_pReturnValue, uint16_t in_ReturnValueLength)

This is the prototype that run functions should follow.

Parameters:

in_BufferCount The number of buffers passed to the run function.

in_ppBufferPointers An array that is in_BufferCount in length that contains the sink side virtual addresses for each buffer passed in to the run function.

in_pBufferLengths An array that is in_BufferCount in length of uint32_t integers describing the length of each passed in buffer in bytes.

in_pMiscData Pointer to the MiscData passed in when the run function was enqueued on the source.

in_MiscDataLen Length in bytes of the MiscData passed in when the run function was enqueued on the source.

in_pReturnValue Pointer to the location where the return value from this run function will be stored.

in_ReturnValueLength Length in bytes of the user-allocated ReturnValue pointer.

Returns:

A uint64_t that can be retrieved in the out_UserData parameter from the COIPipelineWaitForBarrier function.

Definition at line 74 of file COIPipeline_sink.h.

5.17.2 Function Documentation

5.17.2.1 COIRESULT COIPipelineStartExecutingRunFunctions ()

Start processing pipelines on the Sink. This should be done after any required initialization in the Sink's application has finished. No run functions will actually be executed (although they may be queued) until this function is called.

Returns:

COI_SUCCESS if the pipelines were successfully started.

5.18 COIProcessSink

Files

- file COIProcess_sink.h

Functions

- COIRESULT COIProcessWaitForShutdown ()

  This call will block while waiting for the source to send a process destroy message.

5.18.1 Function Documentation

5.18.1.1 COIRESULT COIProcessWaitForShutdown ()

This call will block while waiting for the source to send a process destroy message. This provides the sink side application with a barrier to keep the main() function from exiting until it is directed to by the source. When the shutdown message is received this function will stop any future run functions from executing but will wait for any current run functions to complete. All COI resources will be cleaned up and no additional COI APIs should be called after this function returns. This function does not invoke exit() so the application can perform any of its own cleanup once this call returns.

Returns:

COI_SUCCESS once the process receives the shutdown message.

6 Data Structure Documentation

6.1 COI_ENGINE_INFO Struct Reference

This structure returns information about a MIC engine.

Data Fields

- uint32_t CoreMaxFrequency
  The maximum frequency (in MHz) of the cores on the engine.
- wchar_t DriverVersion [COI_MAX_DRIVER_VERSION_STR_LEN]
  The version string identifying the driver.
- COI_ISA_TYPE ISA
  The ISA supported by the engine.
- uint32_t Load [COI_MAX_HW_THREADS]
  The load percentage for each of the hardware threads on the engine.
- uint32_t NumCores
  The number of cores on the engine.
- uint32_t NumThreads
  The number of hardware threads on the engine.
- uint32_t NumTXS
  The number of texture samplers on the engine.
- uint64_t PhysicalMemory
  The amount of physical memory managed by the OS.
- uint64_t PhysicalMemoryFree
  The amount of free physical memory in the OS.
- uint64_t SwapMemory
  The amount of swap memory managed by the OS.
- uint64_t SwapMemoryFree
  The amount of free swap memory in the OS.

6.1.1 Detailed Description

This structure returns information about a MIC engine. A pointer to this structure is passed into the COIGetEngineInfo() function, which fills in the data before returning to the caller.

Definition at line 49 of file COIEngine_source.h.

6.1.2 Field Documentation

6.1.2.1 uint32_t COI_ENGINE_INFO::CoreMaxFrequency

The maximum frequency (in MHz) of the cores on the engine.

Definition at line 67 of file COIEngine_source.h.

6.1.2.2 wchar_t COI_ENGINE_INFO::DriverVersion[COI_MAX_DRIVER_VERSION_STR_LEN]

The version string identifying the driver.

Definition at line 52 of file COIEngine_source.h.

6.1.2.3 COI_ISA_TYPE COI_ENGINE_INFO::ISA

The ISA supported by the engine.

Definition at line 55 of file COIEngine_source.h.

6.1.2.4 uint32_t COI_ENGINE_INFO::Load[COI_MAX_HW_THREADS]

The load percentage for each of the hardware threads on the engine.

Definition at line 70 of file COIEngine_source.h.

6.1.2.5 uint32_t COI_ENGINE_INFO::NumCores

The number of cores on the engine.

Definition at line 58 of file COIEngine_source.h.

6.1.2.6 uint32_t COI_ENGINE_INFO::NumThreads

The number of hardware threads on the engine.

Definition at line 64 of file COIEngine_source.h.

6.1.2.7 uint32_t COI_ENGINE_INFO::NumTXS

The number of texture samplers on the engine.

Definition at line 61 of file COIEngine_source.h.

6.1.2.8 uint64_t COI_ENGINE_INFO::PhysicalMemory

The amount of physical memory managed by the OS.

Definition at line 73 of file COIEngine_source.h.

6.1.2.9 uint64_t COI_ENGINE_INFO::PhysicalMemoryFree

The amount of free physical memory in the OS.

Definition at line 76 of file COIEngine_source.h.

6.1.2.10 uint64_t COI_ENGINE_INFO::SwapMemory

The amount of swap memory managed by the OS.

Definition at line 79 of file COIEngine_source.h.

6.1.2.11 uint64_t COI_ENGINE_INFO::SwapMemoryFree

The amount of free swap memory in the OS.

Definition at line 82 of file COIEngine_source.h.

6.2 coibarrier Struct Reference

Data Fields

- uint64_t opaque [2]

6.2.1 Detailed Description

Definition at line 26 of file COITypes_common.h.

6.2.2 Field Documentation

6.2.2.1 uint64_t coibarrier::opaque[2]

Definition at line 26 of file COITypes_common.h.

7 File Documentation

7.1 COIBarrier_common.h File Reference

Defines

- #define _COIBARRIER_COMMON_H

Functions

- COIRESULT COIBarrierSignalUserBarrier (COIBARRIER in_Barrier)

  Signal one shot User barrier.

7.1.1 Detailed Description

Definition in file COIBarrier_common.h.

7.1.2 Define Documentation

7.1.2.1 #define _COIBARRIER_COMMON_H

Definition at line 9 of file COIBarrier_common.h.

7.2 COIBarrier_source.h File Reference

Functions

- COIRESULT COIBarrierRegisterUserBarrier (COIBARRIER *out_pBarrier)

  Register a User COIBARRIER so that it can be fired.

- COIRESULT COIBarrierUnregisterUserBarrier (COIBARRIER in_Barrier)

  Unregister a User COIBARRIER.

- COIRESULT COIBarrierWait (uint16_t in_NumBarriers, const COIBARRIER *in_pBarriers, int32_t in_Timeout, uint8_t in_WaitForAll, uint32_t *out_pNumSignaled, uint32_t *out_pSignaledIndices)

  Wait for an arbitrary number of COIBARRIERs to be signaled as completed, eg when the run function or asynchronous map call associated with a barrier has finished execution.

7.2.1 Detailed Description

Definition in file COIBarrier_source.h.

7.3 COIBuffer_sink.h File Reference

Functions

- COIRESULT COIBufferAddRef (void *in_pBuffer)

  Adds a reference to the memory of a buffer.

- COIRESULT COIBufferReleaseRef (void *in_pBuffer)

  Removes a reference to the memory of a buffer.

7.4 COIBuffer_source.h File Reference

Defines

COIBUFFER creation flags.

Please see the COI_VALID_BUFFER_TYPES_AND_FLAGS matrix below which describes the valid combinations of buffer types and flags.

- #define COI_OPTIMIZE_SINK_READ 0x00000010
  Hint to the runtime that the sink will frequently read the buffer.
- #define COI_OPTIMIZE_SINK_WRITE 0x00000020
  Hint to the runtime that the sink will frequently write the buffer.
- #define COI_OPTIMIZE_SOURCE_READ 0x00000004
  Hint to the runtime that the source will frequently read the buffer.
- #define COI_OPTIMIZE_SOURCE_WRITE 0x00000008
  Hint to the runtime that the source will frequently write the buffer.
- #define COI_SAME_ADDRESS_SINKS 0x00000001
  Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated.
- #define COI_SAME_ADDRESS_SINKS_AND_SOURCE 0x00000002
  Create the buffer such that it has the same virtual address on all of the sink processes with which it is associated and in the source process.
- #define COI_SPU_TXS_MEMORY 0x00000040
  Causes the buffer to be allocated from memory that is accessible by the SPU and TXS devices on MIC.

Enumerations

- enum COI_BUFFER_MOVE_FLAG {
  COI_BUFFER_MOVE_DATA = 0,
  COI_BUFFER_NO_MOVE_DATA }
  The buffer move flags are used to indicate when a buffer should be moved when it's state is changed.
- enum COI_BUFFER_STATE {
  COI_BUFFER_VALID = 0,
  COI_BUFFER_INVALID }
  The buffer states are used to indicate whether a buffer is available for access in a COIPROCESS.
- enum COI_BUFFER_TYPE {
  COI_BUFFER_NORMAL = 1,
  COI_BUFFER_STREAMING_TO_SINK,
  COI_BUFFER_STREAMING_TO_SOURCE,
  COI_BUFFER_PINNED }
  The valid buffer types that may be created using COIBufferCreate.

- enum COI_COPY_TYPE {
  COI_COPY_UNSPECIFIED = 0,
  COI_COPY_USE_DMA,
  COI_COPY_USE_CPU }

  This matrix shows the valid combinations of buffer types and map operations that may be passed in to COIBufferMap.

- enum COI_MAP_TYPE {
  COI_MAP_READ_WRITE = 1,
  COI_MAP_READ_ONLY,
  COI_MAP_WRITE_ENTIRE_BUFFER }

  This matrix shows the valid combinations of buffer types and buffer flags that may be passed in to COIBufferCreate and COIBufferCreateFromMemory.

Functions

- COIRESULT COIBufferCopy (COIBUFFER in_DestBuffer, COIBUFFER in_SourceBuffer, uint64_t in_DestOffset, uint64_t in_SourceOffset, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data between two different buffers.

- COIRESULT COIBufferCreate (uint64_t in_Size, COI_BUFFER_TYPE in_Type, uint32_t in_Flags, const void *in_pInitData, uint32_t in_NumProcesses, const COIPROCESS *in_pProcesses, COIBUFFER *out_pBuffer)

  Creates a buffer that can be used in RunFunctions that are queued in pipelines.

- COIRESULT COIBufferCreateFromMemory (uint64_t in_Size, COI_BUFFER_TYPE in_Type, uint32_t in_Flags, void *in_Memory, uint32_t in_NumProcesses, const COIPROCESS *in_pProcesses, COIBUFFER *out_pBuffer)

  Creates a buffer from some existing memory that can be used in RunFunctions that are queued in pipelines.

- COIRESULT COIBufferDestroy (COIBUFFER in_Buffer)

  Destroys a buffer.

- COIRESULT COIBufferGetSinkAddress (COIBUFFER in_Buffer, uint64_t *out_pAddress)

  Gets the Sink's virtual address of the buffer.

- COIRESULT COIBufferMap (COIBUFFER in_Buffer, uint64_t in_Offset, uint64_t in_Length, COI_MAP_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion, COIMAPINSTANCE *out_pMapInstance, void **out_ppData)

  This call initiates a request to access a region of a buffer.

- COIRESULT COIBufferRead (COIBUFFER in_SourceBuffer, uint64_t in_Offset, void *in_pDestData, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data from a buffer into local memory.

- COIRESULT COIBufferSetState (COIBUFFER in_Buffer, COIPROCESS in_Process, COI_BUFFER_STATE in_State, COI_BUFFER_MOVE_FLAG in_DataMove)

  This API allows an experienced COI developer to set where a COIBUFFER is located and when the COIBUFFER's data is moved.

- COIRESULT COIBufferUnmap (COIMAPINSTANCE in_MapInstance, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Disables Source access to the region of the buffer that was provided through the corresponding call to COIBufferMap.

- COIRESULT COIBufferWrite (COIBUFFER in_DestBuffer, uint64_t in_Offset, const void *in_pSourceData, uint64_t in_Length, COI_COPY_TYPE in_Type, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, COIBARRIER *out_pCompletion)

  Copy data from a normal virtual address into an existing COIBUFFER.

7.5 COIEngine_source.h File Reference

Data Structures

- struct COI_ENGINE_INFO

  This structure returns information about a MIC engine.

Defines

- #define COI_MAX_DRIVER_VERSION_STR_LEN 255
- #define COI_MAX_HW_THREADS 1024
- #define COI_MAX_ISA_KNC_DEVICES 32
- #define COI_MAX_ISA_KNF_DEVICES 32
- #define COI_MAX_ISA_x86_64_DEVICES 1

Enumerations

- enum COI_ISA_TYPE {
  COI_ISA_INVALID = 0,
  COI_ISA_x86_64,
  COI_ISA_KNF,
  COI_ISA_KNC }

Functions

- COIRESULT COIEngineGetCount (COI_ISA_TYPE in_ISA, uint32_t *out_pNumEngines)

  Returns the number of engines in the system that match the provided ISA.

- COIRESULT COIEngineGetHandle (COI_ISA_TYPE in_ISA, uint32_t in_EngineIndex, COIENGINE *out_pEngineHandle)

Returns the handle of a user specified engine.

- COIRESULT COIEngineGetInfo (COIENGINE in_EngineHandle, COI_ENGINE_INFO *out_pEngineInfo)

  Returns information related to a specified engine.

7.6 COIMacros_common.h File Reference

Commonly used macros.

Defines

- #define UNREFERENCED_CONST_PARAM(P)
- #define UNREFERENCED_PARAM(P) (P = P)
- #define UNUSED_ATTR __attribute__((unused))

7.6.1 Detailed Description

Commonly used macros.

Definition in file COIMacros_common.h.

7.6.2 Define Documentation

7.6.2.1 #define UNREFERENCED_CONST_PARAM(P)

Value:

```
{ void* x UNUSED_ATTR = \
                                        (void*)(uint64_t)P; \
                                }
```

Definition at line 19 of file COIMacros_common.h.

7.6.2.2 #define UNREFERENCED_PARAM(P) (P = P)

Definition at line 26 of file COIMacros_common.h.

7.6.2.3 #define UNUSED_ATTR __attribute__((unused))

Definition at line 16 of file COIMacros_common.h.

7.7 COIPerf_common.h File Reference

Performance Analysis API.

Functions

- __inline uint64_t COIPerfGetCycleCounter (void)
  Returns a performance counter value.
- uint64_t COIPerfGetCycleFrequency (void)
  Returns the calculated system frequency in hertz.

7.7.1 Detailed Description

Performance Analysis API.

Definition in file COIPerf_common.h.

7.8 COIPipeline_sink.h File Reference

Typedefs

- typedef void(* RunFunctionPtr_t)(uint32_t in_BufferCount, void **in_ppBufferPointers, uint64_t *in_pBufferLengths, void *in_pMiscData, uint16_t in_MiscDataLength, void *in_pReturnValue, uint16_t in_ReturnValueLength)
  This is the prototype that run functions should follow.

Functions

- COIRESULT COIPipelineStartExecutingRunFunctions ()
  Start processing pipelines on the Sink.

7.8.1 Detailed Description

Definition in file COIPipeline_sink.h.

7.9 COIPipeline_source.h File Reference

Defines

- #define COI_PIPELINE_MAX_IN_BUFFERS 32768
- #define COI_PIPELINE_MAX_IN_MISC_DATA_LEN 32768

Enumerations

- enum COI_ACCESS_FLAGS {
  COI_SINK_READ = 1,
  COI_SINK_WRITE,
  COI_SINK_WRITE_ENTIRE }
  These flags specify how a buffer will be used within a run function.

Functions

- COIRESULT COIPipelineClearCPUMask (COI_CPU_MASK *in_Mask)

  Clears a given mask.

- COIRESULT COIPipelineCreate (COIPROCESS in_Process, COI_CPU_MASK in_Mask, uint32_t in_StackSize, COIPIPELINE *out_pPipeline)

  Create a pipeline assoiated with a remote process.

- COIRESULT COIPipelineDestroy (COIPIPELINE in_Pipeline)

  Destroys the inidicated pipeline, releasing its resources.

- COIRESULT COIPipelineFlush (COIPIPELINE in_Pipeline, int32_t in_Timeout)

  Flushes the commands in the pipeline, causing the Sink to start processing run functions from the indicated pipeline, and waits for all run functions that have been queued on the pipeline before the flush to finish before the call returns.

- COIRESULT COIPipelineGetEngine (COIPIPELINE in_Pipeline, COIENGINE *out_pEngine)

  Retrieve the engine that the pipeline is associated with.

- COIRESULT COIPipelineRunFunction (COIPIPELINE in_Pipeline, COIFUNCTION in_Function, uint32_t in_NumBuffers, const COIBUFFER *in_Buffers, const COI_ACCESS_FLAGS *in_pBufferAccessFlags, uint32_t in_NumDependencies, const COIBARRIER *in_pDependencies, const void *in_pMiscData, uint16_t in_MiscDataLen, void *out_pAsyncReturnValue, uint16_t in_AsyncReturnValueLen, COIBARRIER *out_pCompletion)

  Enqueues a function in the remote process binary to be executed.

- COIRESULT COIPipelineSetCPUMask (COIPROCESS in_Process, uint32_t in_CoreID, uint8_t in_ThreadID, COI_CPU_MASK *out_pMask)

  Set a given mask to a particular core:thread pair.

7.9.1 Detailed Description

Definition in file COIPipeline_source.h.

7.10 COIProcess_sink.h File Reference

Functions

- COIRESULT COIProcessWaitForShutdown ()

  This call will block while waiting for the source to send a process destroy message.

7.10.1 Detailed Description

Definition in file COIProcess_sink.h.

7.11 COIProcess_source.h File Reference

Defines

- #define COI_MAX_FILE_NAME_LENGTH 256
- #define COI_MAX_FUNCTION_NAME_LENGTH 256
- #define COI_PROCESS_SOURCE ((COIPROCESS)-1)

  This is a special COIPROCESS handle that can be used to indicate that the source process should be used for an operation.

Enumerations

- enum COI_SHUTDOWN_REASON {
  COI_SHUTDOWN_OK = 0,
  COI_SHUTDOWN_SIGTERM,
  COI_SHUTDOWN_SEGFAULT }

Functions

- COIRESULT COIProcessCreateFromFile (COIENGINE in_Engine, const char *in_pBinaryName, int in_Argc, const char **in_ppArgv, uint8_t in_DupEnv, const char **in_ppAdditionalEnv, uint8_t in_ProxyActive, const char *in_ProxyRoot, uint64_t in_BufferSpace, COIPROCESS *out_pProcess)

  Create a remote process on the Sink and start executing it's main() function.

- COIRESULT COIProcessCreateFromMemory (COIENGINE in_Engine, const char *in_pBinaryName, const void *in_pBinaryBuffer, uint64_t in_BinaryBufferLength, int in_Argc, const char **in_ppArgv, uint8_t in_DupEnv, const char **in_ppAdditionalEnv, uint8_t in_ProxyActive, const char *in_ProxyRoot, uint64_t in_BufferSpace, COIPROCESS *out_pProcess)

  Create a remote process on the Sink and start executing it's main() function.

- COIRESULT COIProcessDestroy (COIPROCESS in_Process, int32_t in_WaitForMainTimeout, uint8_t in_ForceDestroy, int8_t *out_pProcessReturn, COI_SHUTDOWN_REASON *out_pReason)

  Destroys the indicated process, releasing its resources.

- COIRESULT COIProcessGetFunctionHandles (COIPROCESS in_Process, uint32_t in_NumFunctions, const char **in_ppFunctionNameArray, COIFUNCTION *out_pFunctionHandleArray)

  Given a loaded native process, gets an array of function handles that can be used to schedule run functions on a pipeline associated with that process.

- COIRESULT COIProcessLoadLibraryFromFile (COIPROCESS in_Process, const char *in_pFileName, const char *in_pLibraryName, COILIBRARY *out_pLibrary)

  Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows.

- COIRESULT COIProcessLoadLibraryFromMemory (COIPROCESS in_Process, const void *in_pLibraryBuffer, uint64_t in_LibraryBufferLength, const char *in_pLibraryName, COILIBRARY *out_pLibrary)

Loads a shared library into the specified remote process, akin to using dlopen() on a local process in Linux or LoadLibrary() in Windows.

- COIRESULT COIProcessUnLoadLibrary (COIPROCESS in_Process, COILIBRARY in_Library)

  Unloads a a previously loaded shared library from the specified remote process.

7.11.1 Detailed Description

Definition in file COIProcess_source.h.

7.12 COIResult_common.h File Reference

Enumerations

- enum COIRESULT {
  COI_SUCCESS = 0,
  COI_ERROR,
  COI_NOT_INITIALIZED,
  COI_ALREADY_INITIALIZED,
  COI_ALREADY_EXISTS,
  COI_DOES_NOT_EXIST,
  COI_INVALID_POINTER,
  COI_OUT_OF_RANGE,
  COI_NOT_SUPPORTED,
  COI_TIME_OUT_REACHED,
  COI_DUPLICATE_OBJECT,
  COI_ARGUMENT_MISMATCH,
  COI_SIZE_MISMATCH,
  COI_OUT_OF_MEMORY,
  COI_INVALID_HANDLE,
  COI_RETRY,
  COI_RESOURCE_EXHAUSTED,
  COI_ALREADY_LOCKED,
  COI_NOT_LOCKED,
  COI_MISSING_DEPENDENCY,
  COI_UNDEFINED_SYMBOL,
  COI_PENDING,
  COI_BINARY_AND_HARDWARE_MISMATCH,
  COI_PROCESS_DIED,
  COI_INVALID_FILE,
  COI_BARRIER_CANCELED,
  COI_NUM_RESULTS }

Functions

- const char * COIResultGetName (COIRESULT in_ResultCode)

  Returns the string version of the passed in COIRESULT.

Variables

- *If you see an error on this line

7.12.1 Variable Documentation

7.12.1.1 * If you see an error on this line

Definition at line 212 of file COIResult_common.h.

7.13 COISysInfo_common.h File Reference

This interface allows developers to query the platform for system level information.

Defines

- #define INITIAL_APIC_ID_BITS 0xFF000000
- #define NUMBER_HW_THREADS 128

Functions

- uint32_t COISysGetAPICID (void)
- uint32_t COISysGetCoreCount (void)
- uint32_t COISysGetCoreIndex (void)
- uint32_t COISysGetHardwareThreadCount (void)
- uint32_t COISysGetHardwareThreadIndex (void)
- uint32_t COISysGetL2CacheCount (void)
- uint32_t COISysGetL2CacheIndex (void)

7.13.1 Detailed Description

This interface allows developers to query the platform for system level information.

Definition in file COISysInfo_common.h.

7.14 COITypes_common.h File Reference

Data Structures

- struct coibarrier

Typedefs

- typedef uint32_t COI_CPU_MASK [8]
- typedef struct coibarrier COIBARRIER
- typedef struct coibuffer * COIBUFFER
- typedef struct coiengine * COIENGINE
- typedef struct coifunction * COIFUNCTION
- typedef struct coilibrary * COILIBRARY
- typedef struct coimapinst * COIMAPINSTANCE
- typedef struct coipipeline * COIPIPELINE
- typedef struct coiprocess * COIPROCESS

7.14.1 Detailed Description

Definition in file COITypes_common.h.

Index