oneAPI Level Zero Specification

Intel

May 28, 2020

OVERVIEW

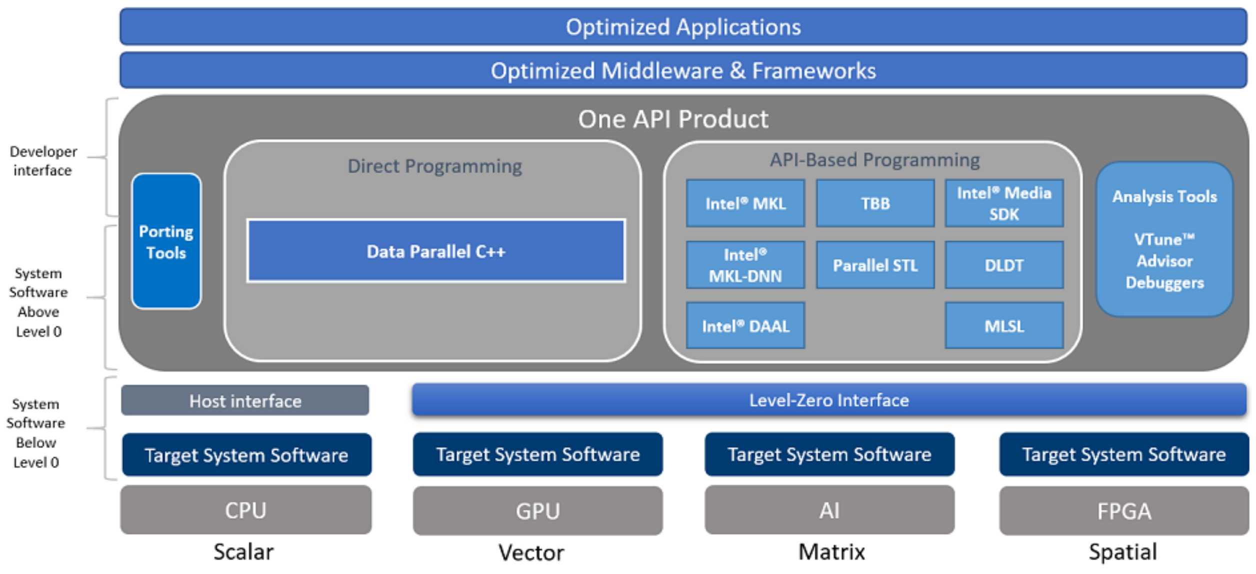
CHAPTER

ONE

OVERVIEW

1.1 Objective

The objective of the 'One API' Level-Zero API is to provide direct-to-metal interfaces to offload accelerator devices. It is a programming interface that can be published at a cadence that better matches Intel hardware releases and can be tailored to any device needs. It can be adapted to support broader set of languages features, such as function pointers, virtual functions, unified memory, and I/O capabilities.



The Level-Zero API provides the lowest-level, fine-grain and most explicit control over:

- Device Discovery
- Memory Allocation
- Peer-to-Peer Communication
- Inter-Process Sharing
- Kernel Submission
- Asynchronous Execution and Scheduling
- Synchronization Primitives
- Metrics Reporting
- System Management Interface

Most applications should not require the additional control provided by the Level-Zero API. The Level-Zero API is intended for providing explicit controls needed by higher-level runtime APIs and libraries.

While initially influenced by other low-level APIs, such as OpenCL and Vulkan, the Level-Zero APIs are designed to evolve independently. While initially influenced by GPU architecture, the Level-Zero APIs are designed to be supportable across different compute device architectures, such as FPGAs, CSAs, etc.

1.1.1 Devices

The API architecture exposes both physical and logical abstraction of the underlying devices capabilities. The device, sub device and memory are exposed at physical level while command queues, events and synchronization methods are defined as logical entities. All logical entities will be bound to device level physical capabilities.

Device discovery APIs enumerate the accelerators functional features. These APIs provide interface to query information like compute unit count within the device or sub device, available memory and affinity to the compute, user managed cache size and work submission command queues.

1.1.2 Memory & Caches

Memory is visible to the upper-level software stack as unified memory with a single virtual address space covering both the Host and a specific device.

For GPUs, the API exposes two levels of the device memory hierarchy:

1. Local Device Memory: can be managed at the device and/or sub device level.
2. Device Cache(s):
    - Last Level Cache (L3) can be controlled through memory allocation APIs.
    - Low Level Cache (L1) can be controlled through program language intrinsics.

The API allows allocation of buffers and images at device and sub device granularity with full cacheablity hints.

- Buffers are transparent memory accessed through virtual address pointers
- Images are opaque objects accessed through handles

The memory APIs provide allocation methods to allocate either device, host or shared memory. The APIs enable both implicit and explicit management of the resources by the application or runtimes. The interface also provides query capabilities for all memory objects.

1.1.3 Subdevice Support

The API supports sub-devices and there are functions to query and obtain a sub-device. A sub-device can represent a physical or logical partition of the device. Outside of these functions there are no distinction between sub-devices and devices. For example, a sub-device can be used with memory allocation and tasks and allow placement and submission to a specific sub-device.

1.1.4 Peer-to-Peer Communication

Peer to Peer API's provide capabilities to marshall data across Host to Device, Device to Host and Device to Device. The data marshalling API can be scheduled as asynchronous operations or can be synchronized with kernel execution through command queues. Data coherency is maintained by the driver without any explicit involvement from the application.

1.1.5 Inter-Process Communication

The API allows sharing of memory objects across different device processes. Since each process has its own virtual address space, there is no guarantee that the same virtual address will be available when the memory object is shared in new process. There are a set of APIs that makes it easier to share the memory objects with ease.

1.1.6 System Management

The API provides in-band ability to query the performance, power and health of accelerator resources. It also enables controlling the performance and power profile of these resources. Finally, it provides access to maintenance facilities such as performing hardware diagnostics or updating firmware.

1.2 API Specification

The following section provides high-level design philosophy of the APIs. For more detailed information, refer to the programming guides and detailed specification pages.

1.2.1 Terminology

This specification uses key words based on RFC2119 to indicate requirement level. In particular, the following words are used to describe the actions of an implementation of this specification:

- **May** - the word *may*, or the adjective *optional*, mean that conforming implementations are permitted to, but need not behave as described.
- **Should** - the word *should*, or the adjective *recommended*, mean that there could be reasons for an implementations to deviate from the behavior described, but that such deviation should be avoided.
- **Must** - the word *must*, or the term *required* or *shall*, mean that the behavior described is an absolute requirement of the specification.

1.2.2 Naming Convention

The following naming conventions are followed in order to avoid conflicts within the API, or with other APIs and libraries: - all driver entry points are prefixed with ze - all types follow **ze_<name>_t** convention - all macros and enumerator values use all caps **ZE_<SCOPE>_<NAME>** convention - all functions use camel case **ze<Object><Action>** convention - exception: since "driver" functions use implicit <Object>, it is omitted - all structure members and function parameters use camel case convention

In addition, the following coding standards are followed: - all function input parameters precede output parameters - all functions return ::ze_result_t

1.2.3 Versioning

There are multiple versions that should be used by the application to determine compatibility:

API Version - this is the version of the API supported by the device.

- This is typically used to determine if the device supports the minimum set of APIs required by the application.
- There is a single API version that represents a collection of APIs.
- The value is determined from calling ::zeDriverGetApiVersion
- The value returned will be the minimum of the ::ze_api_version_t supported by the device and known by the driver.

Structure Version - these are the versions of the structures passed-by-pointer to the driver.

- These are typically used by the driver to support applications written to older versions of the API.
- They are provided as the first member of every structure passed to the driver.

Driver Version - this is the version of the driver installed in the system.

- This is typically used to mitigate driver implementation issues for a feature.
- The value is determined from calling ::zeDriverGetProperties

1.2.4 Error Handling

The following design philosophies are adopted in order to reduce Host-side overhead:

- By default, the driver implementation does no parameter validation of any kind
  - This can be enabled via environment variables, described below
- By default, neither the driver nor device provide any protection against the following:
  - Invalid API programming
  - Invalid function arguments
  - Function infinite loops or recursions
  - Synchronization primitive deadlocks
  - Non-visible memory access by the Host or device
  - Non-resident memory access by the device
- The driver implementation is **not** required to perform API validation of any kind
  - The driver should ensure well-behaved applications are not burdened with the overhead needed for non-behaving applications
  - Unless otherwise specified, the driver behavior is undefined when APIs are improperly used
  - For debug purposes, API validation can be enabled via the [Validation Layers](#v0)
- All API functions return ::ze_result_t
  - This enumeration contains error codes for the core APIs and validation layers
  - This allows for a consistent pattern on the application side for catching errors; especially when validation layer(s) are enabled

1.2.5 Multithreading and Concurrency

The following design philosophies are adopted in order to maximize Host thread concurrency:

- APIs are free-threaded when the driver object handle is different.
  - the driver should avoid thread-locks for these API calls
- APIs are not thread-safe when the driver object handle is the same, except when explicitly noted.
  - the application is responsible for ensuring multiple threads do not enter an API when the handle is the same
- APIs are not thread-safe with other APIs that use the same driver object handle
  - the application is responsible for ensuring multiple threads do not enter these APIs when the handle is the same
- APIs do not support reference counting of handles.
  - the application is responsible for tracking ownership and explicitly freeing handles and memory
  - the application is responsible for ensuring that all driver objects and memory are no longer in-use by the device before freeing; otherwise the Host or device may fault
  - no implicit garabage collection is supported by the driver

In general, the API is designed to be free-threaded rather than thread-safe. This provides multithreaded applications with complete control over both threading and locks. This also eliminates unnecessary driver overhead for single threaded applications and/or very low latency usages.

The exception to this rule is that all memory allocation APIs are thread-safe since they allocate from a single global memory pool. If an application needs lock-free memory allocation, then it could allocate a per-thread pool and implement its own sub-allocator.

An application is in direct control over all Host thread creation and usage. The driver should never implicitly create threads. If there is a need for an implementation to use a background thread, then that thread should be create and provided by the application.

Each API function must document details on the multithreading requirements for that call.

The primary usage-models enabled by these rules is:

- multiple, simultaneous threads may operate on independent driver objects with no implicit thread-locks
- driver object handles may be passed between and used by multiple threads with no implicit thread-locks

1.2.6 Experimental API Support

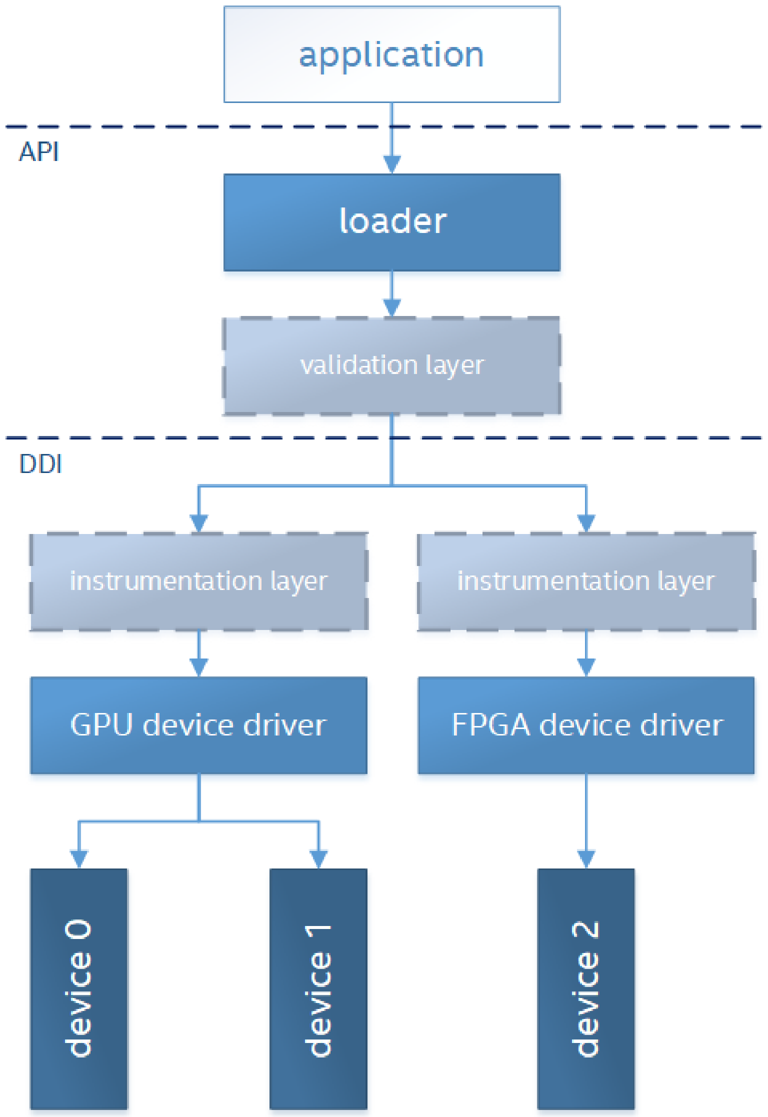
Features which are still being considered for inclusion into the "Core" API, but require additional experimentation by application vendors before ratification, are exposed as "Experimental" APIs.

Applications should not rely on experimental APIs in production. - Experimental APIs may be added and removed from the API at any time; with or without an official API revision. - Experimental APIs are not guaranteed to be forward or backward compatible between API versions. - Experimental APIs are not guaranteed to be supported in production driver releases; and may appear and disappear from release to release.

An implementation will return ::ZE_RESULT_ERROR_UNSUPPORTED_FEATURE for any experimental API not supported by that driver.

1.3 Driver Architecture

The following section provides high-level driver architecture.



1.3.1 Library

A static import library shall be provided to allow applications to make direct API calls without understanding the underlying driver interfaces.

C/C++ applications may include "ze_api.h" and link with "ze_api.lib".

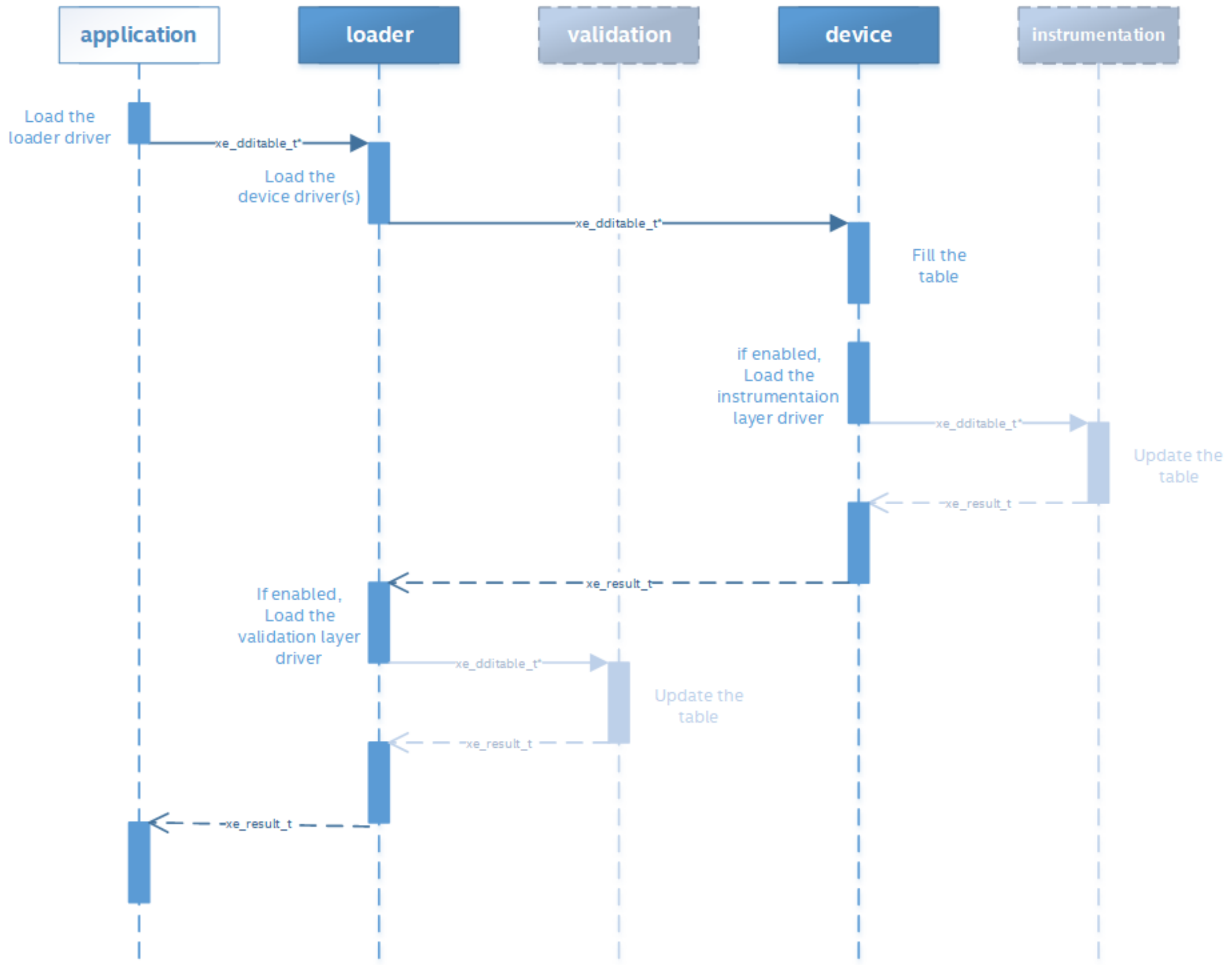
1.3.2 Loader

The loader initiates the loading of the driver(s) and layer(s). The loader exports all API functions to the static library via per-process API function pointer table(s). Each driver and layer must below the loader will also export its API/DDI functions via per-process function pointer table(s). The export function and table definitions are defined in "ze_ddi.h".

The loader is dynamically linked with the application using the "ze_loader.dll" (windows) or "ze_loader.so" (linux). The loader is vendor agnostic, but must be aware of the names of vendor-specific device driver names. (Note: these are currently hard-coded but a registration method will be adopted when multiple vendors are supported.)

The loader dynamically loads each vendor's device driver(s) present in the system and queries each per-process function pointer table(s). If only one device driver needs to be loaded, then the loader layer may be entirely bypassed.

The following diagram illustrates the expected loading sequence:



Thus, the loader's internal function pointer table entries may point to:

- validation layer intercepts (if enabled),
- instrumentation layer intercepts (if enabled),
- device driver exports, + or any combination of the above

1.3.3 Device Drivers

The device driver(s) contain the device-specific implementations of the APIs.

The device driver(s) are dynamically linked using a *ze_vendor_type.dll* (windows) / *ze_vendor_type.so* (linux); where *vendor* and *type* are names chosen by the device vendor. For example, Intel GPUs use the name: "ze_intc_gpu".

1.3.4 Validation Layer

The validation layer provides an optional capability for application developers to enable additional API validation while maintaining minimal driver implementation overhead.

- works independent of driver implementation
- works for production / release drivers
- works independent of vendor or device type
- checks for common application errors, such as parameter validation
- provides common application debug tracking, such as object and memory lifetime

The validation layer must be enabled via an environment variable. Each capability is enabled by additional environment variables.

The validation layer supports the following capabilities:

- Parameter Validation
  - checks function parameters, such as null pointer parameters, invalid enumerations, uninitialized structures, etc.
- Handle Lifetime
  - tracks handle allocations, destruction and usage for leaks and invalid usage (e.g., destruction while still in-use by device)
- Memory Tracker
  - tracks memory allocations and free for leaks and invalid usage (e.g., non-visible to device)
- Threading Validation
  - checks multi-threading usage (e.g., functions are not called from simultaneous threads using the same handle)

1.3.5 Instrumentation Layer

The instrumentation layer provides an optional capability for application developers to enable additional profiling API while maintaining minimal driver implementation overhead.

- works independent of driver implementation
- works for production / release drivers
- implements *Tools* APIs

The instrumentation layer must be enabled via an environment variable. Each capability is enabled by additional environment variables.

The instrumentation layer supports the following capabilities:

- **API Tracing**
  - Enables API tracing and profiling APIs; more details in Tools programming guide
- **Program Instrumentation**
  - Enables instrumentation of programs for profiling; more details in Tools programming guide

1.3.6 Environment Variables

The following table documents the supported knobs for overriding default driver behavior.

Category	Name	Values	Description
Device	ZE_AFFINITY_MASK	Hex String	Forces driver to only report devices (and sub-devices) as specified by mask value
Memory	ZE_SHARED_FORCE_DEVICE_ALLOC	{**0**, 1}	Forces all shared allocations into device memory
Validation	ZE_ENABLE_VALIDATION_LAYER	{**0**, 1}	Enables validation layer for debugging
	ZE_ENABLE_PARAMETER_VALIDATION	{**0**, 1}	Enables the validation level for parameters
	ZE_ENABLE_HANDLE_LIFETIME	{**0**, 1}	Enables the validation level for tracking handle lifetime
	ZE_ENABLE_MEMORY_TRACKER	{**0**, 1}	Enables the validation level for tracking memory lifetime
	ZE_ENABLE_THREADING_VALIDATION	{**0**, 1}	Enables the validation level for multithreading usage
Instrumentation	ZE_ENABLE_INSTRUMENTATION_LAYER	{**0**, 1}	Enables validation layer for debugging
	ZE_ENABLE_API_TRACING	{**0**, 1}	Enables the instrumentation for API tracing
	ZE_ENABLE_METRICS	{**0**, 1}	Enables the instrumentation for device metrics
	ZE_ENABLE_PROGRAM_INSTRUMENTATION	{**0**, 1}	Enables the instrumentation for program instrumentation
	ZE_ENABLE_PROGRAM_DEBUGGING	{**0**, 1}	Enables the instrumentation for program debugging

Affinity Mask

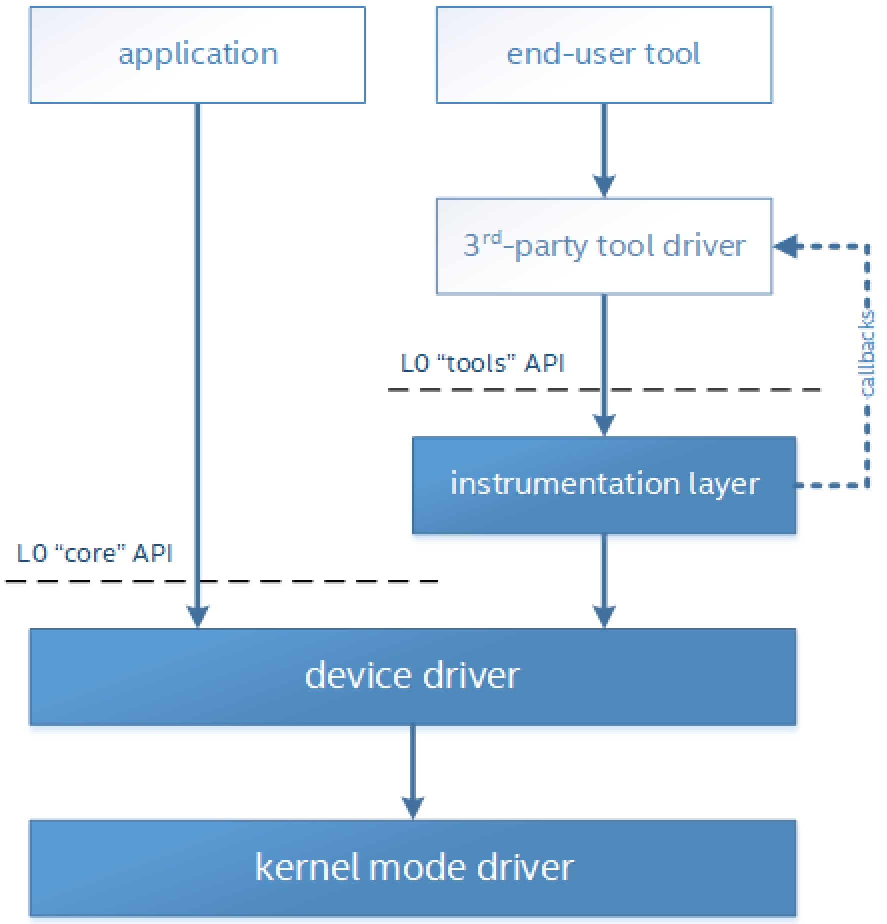
The affinity mask allows an application or tool to restrict which devices (and sub-devices) are visible to 3rd-party libraries or applications in another process, respectively. The affinity mask is specified via an environment variable as a string of hexadecimal values. The value is specific to system configuration; e.g., the number of devices and the number of sub-devices for each device. The following examples demonstrate proper usage:

- "" (empty string) = disabled; i.e. all devices and sub-devices are reported. This is the default value.
- Two devices, each with four sub-devices
  - "FF" = all devices and sub-devices are reported (same as default)
  - "0F" = only device 0 (with all its sub-devices) is reported
  - "F0" = only device 1 (with all its sub-devices) is reported as device 0'
  - "AA" = both device 0 and 1 are reported, however each only has two sub-devices reported as sub-device 0 and 1
- Two devices, device 0 with one sub-device and device 1 with two sub-devices
  - "07" = all devices and sub-devices are reported (same as default) + "01" = only device 0 (with all its sub-devices) is reported
  - "06" = only device 1 (with all its sub-devices) is reported as device 0
  - "05" = both device 0 and device 1 are reported, however each only has one sub-device reported as sub-device 0

1.4 Tools

Level-Zero APIs specific for supporting 3rd-party tools are separated from "Core" into "Tools" APIs. The "Tools" APIs are designed to provided low-level access to device capabilities in order to support 3rd-party tools, but are not intended to replace or directly interface 3rd-party tools. The "Tools" APIs are still available for direct application use.

The following diagram illustrates how 3rd-party tools may utilize the instrumentation layer:



The "Tools" APIs provide the following capabilities for 3rd-party tools:

- Allow for callbacks to be registered, in order to be notified of specific application events.
- Allow for device metrics to be queried, in order to profile application usage.
- Allow for application programs to be instrumented with custom instructions, for low-level code profiling.
- Allow for application programs to be debugged using breakpoints and register access.

See the "Tools" programming guide for more details.

1.5 System Management

All global management of accelerator resources are separated from "Core" into the "Sysman" API.

The "Sysman" API provides in-band access to the following features for each accelerator device:

- Query inventory information
- Query information about host processes using the device
- Change the accelerator workload scheduling policies
- Query and control frequency/voltage/power
- Query temperature sensors

- Query load on various accelerator engines (overall, media, compute, copy)
- Query device memory bandwidth and health
- Query PCI bandwidth and health
- Query high-speed Fabric bandwidth and health
- Control the standby policy of the device
- Query ECC/RAS status of various components on the device
- Query power supply status
- Control LEDs
- Control fans
- Perform overclocking/under-voltage changes where appropriate
- Listen for events (temperature excursion, frequency throttling, RAS errors)
- Flash firmware
- Run diagnostics
- Reset the device

By default, only administrator users have permissions to perform control operations on resources. Most queries are available to any user with the exception of those that could be used for side-channel attacks. The systems administrator can tighten/relax the default permissions.

See the “Sysman” programming guide for more details.

CHAPTER

TWO

CORE PROGRAMMING GUIDE

2.1 Drivers and Devices

2.1.1 Drivers

A driver represents a collection of physical devices in the system using the same Level-Zero driver.

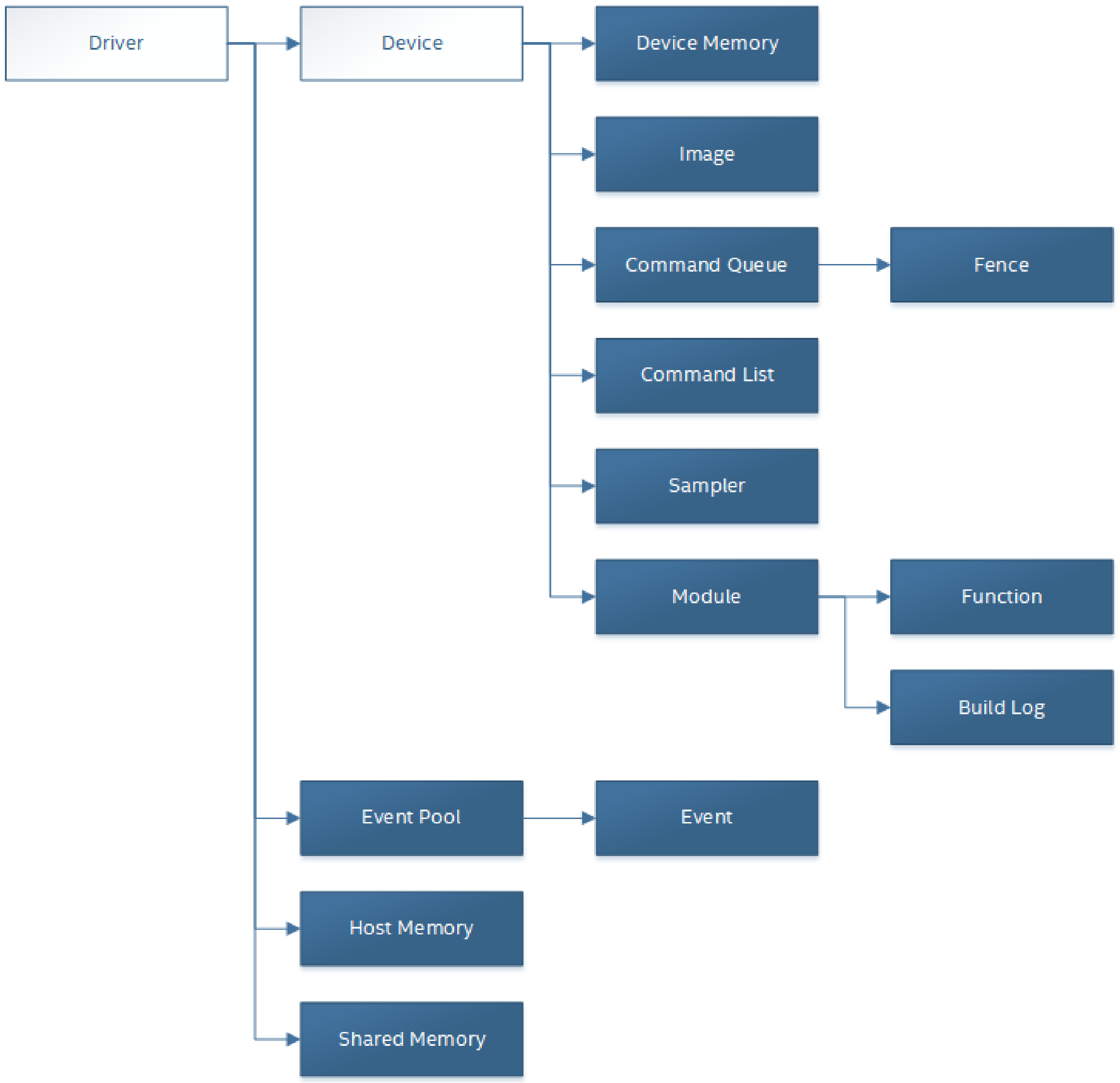
- The application may query the number of Level-Zero drivers installed on the system and the properties of each driver.
- More than one driver may be available in the system. For example, one driver may support two GPUs from one vendor, another driver supports a GPU from a different vendor, and finally a different driver may support an FPGA.
- A driver is primarily used to allocate and manage resources that are used by multiple devices.
- Memory is **not** implicitly shared across all devices supported by a driver. However, it is available to be explicitly shared.

2.1.2 Device

A device represents a physical device in the system that support Level-Zero.

- The application may query the number devices supported by a driver.
- The application is responsible for sharing memory and explicit submission and synchronization across multiple devices.
- Device may expose sub-devices that allow finer-grained control of physical or logical partitions of a device.

The following diagram illustrates the relationship between the driver, device and other objects described in this document.



2.1.3 Initialization and Discovery

The driver must be initialized by calling Init before any other function. This function will load and initialize all Level-Zero driver(s) in the system for all threads in the current process. Simultaneous calls to ::zeInit are thread-safe and only one instance of driver(s) will be loaded per-process. This function will allow queries of the available driver instances in the system.

The following pseudo-code demonstrates a basic initialization and device discovery sequence:

```
// Initialize the driver
zeInit(ZE_INIT_FLAG_NONE);

// Discover all the driver instances
uint32_t driverCount = 0;
```

(continues on next page)

(continued from previous page)

```
zeDriverGet(&driverCount, nullptr);

ze_driver_handle_t* allDrivers = allocate(driverCount * sizeof(ze_driver_handle_t));
zeDriverGet(&driverCount, allDrivers);

// Find a driver instance with a GPU device
ze_driver_handle_t hDriver = nullptr;
ze_device_handle_t hDevice = nullptr;
for(i = 0; i < driverCount; ++i) {
    uint32_t deviceCount = 0;
    zeDeviceGet(allDrivers[i], &deviceCount, nullptr);

    ze_device_handle_t* allDevices = allocate(deviceCount * sizeof(ze_device_handle_
↪t));
    zeDeviceGet(allDrivers[i], &deviceCount, allDevices);

    for(d = 0; d < deviceCount; ++d) {
        ze_device_properties_t device_properties;
        zeDeviceGetProperties(allDevices[d], &device_properties);

        if(ZE_DEVICE_TYPE_GPU == device_properties.type) {
            hDriver = allDrivers[i];
            hDevice = allDevices[d];
            break;
        }
    }

    free(allDevices);
    if(nullptr != hDriver) {
        break;
    }
}

free(allDrivers);
if(nullptr == hDevice)
    return; // no GPU devices found

...
```

2.2 Memory and Images

There are two types of allocations:

1. *Memory* - linear, unformatted allocations for direct access from both the host and device.
2. *Images* - non-linear, formatted allocations for direct access from the device.

2.2.1 Memory

Linear, unformatted memory allocations are represented as pointers in the host application. A pointer on the host has the same size as a pointer on the device.

Types

Three types of allocations are supported. The type of allocation describes the *ownership* of the allocation:

1. **Host** allocations are owned by the host and are intended to be allocated out of system memory. Host allocations are accessible by the host and one or more devices. The same pointer to a host allocation may be used on the host and all supported devices; they have *address equivalence*. Host allocations are not expected to migrate between system memory and device local memory. Host allocations trade off wide accessibility and transfer benefits for potentially higher per-access costs, such as over PCI express.

2. **Device** allocations are owned by a specific device and are intended to be allocated out of device local memory, if present. Device allocations generally trade off access limitations for higher performance. With very few exceptions, device allocations may only be accessed by the specific device that they are allocated on, or copied to a host or another device allocation. The same pointer to a device allocation may be used on any supported device.

3. **Shared** allocations share ownership and are intended to migrate between the host and one or more devices. Shared allocations are accessible by at least the host and an associated device. Shared allocations may be accessed by other devices in some cases. Shared allocations trade off transfer costs for per-access benefits. The same pointer to a shared allocation may be used on the host and all supported devices.

A **Shared System** allocation is a sub-class of a **Shared** allocation, where the memory is allocated by a *system allocator* - such as `malloc` or `new` - rather than by an allocation API. Shared system allocations have no associated device - they are inherently cross-device. Like other shared allocations, shared system allocations are intended to migrate between the host and supported devices, and the same pointer to a shared system allocation may be used on the host and all supported devices.

In summary:

Name	Initial Location	Accessible By		Migratable To	
Host	Host	Host	Yes	Host	N/A
		Any Device	Yes (perhaps over PCIe)	Device	No
Device	Specific Device	Host	No	Host	No
		Specific Device	Yes	Device	N/A
		Another Device	Optional (may require p2p)	Another Device	No
Shared	Host, Specific Device, or Unspecified	Host	Yes	Host	Yes
		Specific Device	Yes	Device	Yes
		Another Device	Optional (may require p2p)	Another Device	Optional
Shared System	Host	Host	Yes	Host	Yes
		Device	Yes	Device	Yes

Devices may support different capabilities for each type of allocation. Supported capabilities are:

- ::ZE_MEMORY_ACCESS - if a device supports access (read or write) to allocations of the specified type.

- ::ZE_MEMORY_ATOMIC_ACCESS - if a device support atomic operations on allocations of the specified type. Atomic operations may include relaxed consistency read-modify-write atomics and atomic operations that enforce memory consistency for non-atomic operations.
- ::ZE_MEMORY_CONCURRENT_ACCESS - if a device supports concurrent access to allocations of the specified type. Concurrent access may be from another device that supports concurrent access, or from the host. Devices that support concurrent access but do not support concurrent atomic access must write to unique non-overlapping memory locations to avoid data races and hence undefined behavior.
- ::ZE_MEMORY_CONCURRENT_ATOMIC_ACCESS - if a device supports concurrent atomic operations on allocations of the specified type. Concurrent atomic operations may be from another device that supports concurrent atomic access, or from the host. Devices that support concurrent atomic access may use atomic operations to enforce memory consistency with other devices that support concurrent atomic access, or with the host.

Some devices may *oversubscribe* some **shared** allocations. When and how such oversubscription occurs, including which allocations are evicted when the working set changes, are considered implementation details.

The required matrix of capabilities are:

Allocation Type	Access	Atomic Access	Concurrent Access	Concurrent Atomic Access
Host	Required	Optional	Optional	Optional
Device	Required	Optional	Optional	Optional
Shared	Required	Optional	Optional	Optional
Shared (Cross-Device)	Optional	Optional	Optional	Optional
Shared System (Cross-Device)	Optional	Optional	Optional	Optional

Cache Hints, Prefetch, and Memory Advice

Cacheability hints may be provided via separate host and device allocation flags when memory is allocated.

Shared allocations may be prefetched to a supporting device via the ::zeCommandListAppendMemoryPrefetch API. Prefetching may allow memory transfers to be scheduled concurrently with other computations and may improve performance.

Additionally, an application may provide memory advice for a **shared** allocation via the ::zeCommandListAppendMemAdvise API, to override driver heuristics or migration policies. Memory advice may avoid unnecessary or unprofitable memory transfers and may improve performance.

Both prefetch and memory advice are asynchronous operations that are appended into command lists.

2.2.2 Images

An image is used to store multi-dimensional and format-defined memory for optimal device access. An image's contents can be copied to and from other images, as well as host-accessible memory allocations. This is the only method for host access to the contents of an image. This methodology allows for device-specific encoding of image contents (e.g., tile swizzle patterns, lossless compression, etc.) and avoids exposing these details in the API in a backwards compatible fashion.

```
// Specify single component FLOAT32 format
ze_image_format_desc_t formatDesc = {
    ZE_IMAGE_FORMAT_LAYOUT_32, ZE_IMAGE_FORMAT_TYPE_FLOAT,
    ZE_IMAGE_FORMAT_SWIZZLE_R, ZE_IMAGE_FORMAT_SWIZZLE_0, ZE_IMAGE_FORMAT_SWIZZLE_0,␣
↪ZE_IMAGE_FORMAT_SWIZZLE_1
};

ze_image_desc_t imageDesc = {
    ZE_IMAGE_DESC_VERSION_CURRENT,
    ZE_IMAGE_FLAG_PROGRAM_READ,
    ZE_IMAGE_TYPE_2D,
    formatDesc,
    128, 128, 0, 0, 0
};
ze_image_handle_t hImage;
zeImageCreate(hDevice, &imageDesc, &hImage);

// upload contents from host pointer
zeCommandListAppendImageCopyFromMemory(hCommandList, hImage, nullptr, pImageData,␣
↪nullptr);
...
```

2.2.3 Device Cache Settings

There are two methods for device and kernel cache control:

1. Cache Size Configuration: Ability to configure larger size for SLM vs Data globally for Device or per Kernel instance.
2. Runtime Hint/preference for application to allow access to be Cached or not in Device Caches. For GPU device this is provided via two ways:
    - During Image creation via Flag
    - Kernel instruction

The following pseudo-code demonstrates a basic sequence for Cache size configuration:

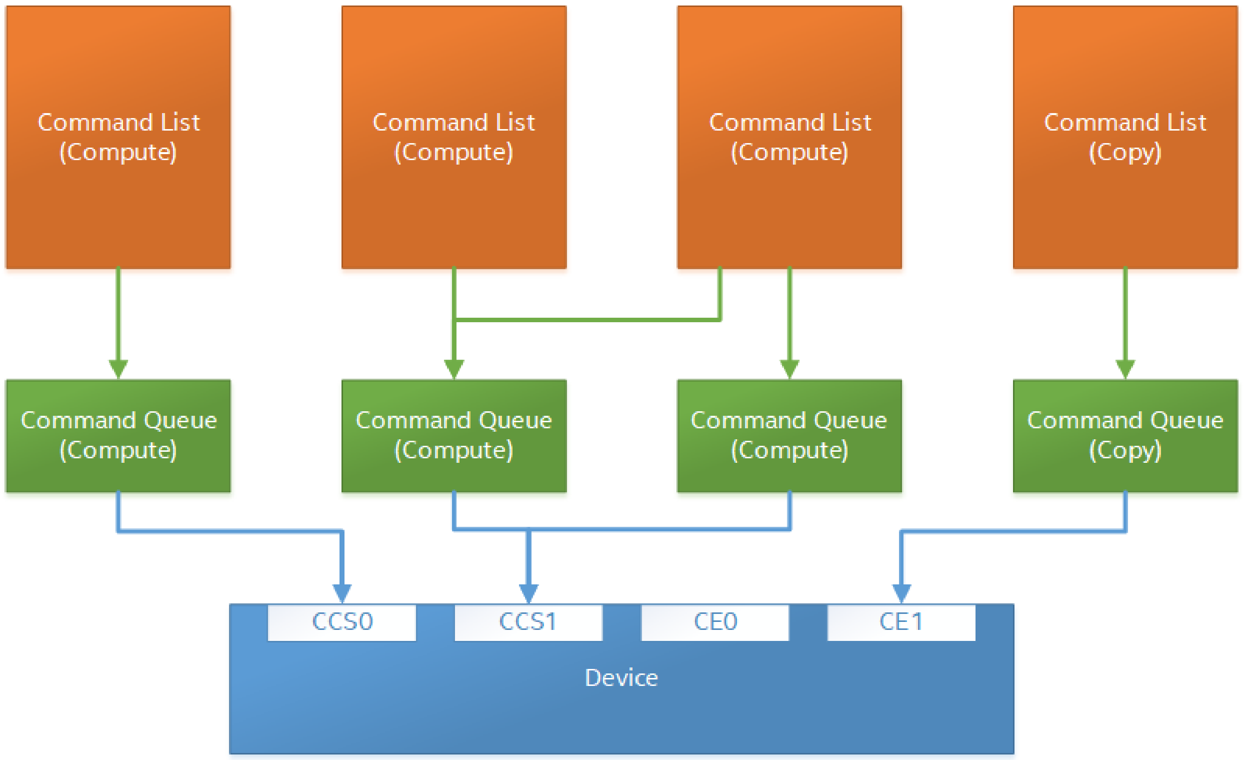
```
// Large SLM for Intermediate and Last Level cache.
// Note: The intermediate cache setting is applied to each kernel. Last level is␣
↪applied for the device.
zeKernelSetIntermediateCacheConfig(hKernel, ZE_CACHE_CONFIG_LARGE_SLM);
zeDeviceSetLastLevelCacheConfig(hDevice, ZE_CACHE_CONFIG_LARGE_SLM);
...
```

2.3 Command Queues and Command Lists

The following are the motivations for separating a command queue from a command list:

- Command queues are mostly associated with physical device properties, such as the number of input streams.
- Command queues provide (near) zero-latency access to the device.
- Command lists are mostly associated with Host threads for simultaneous construction.
- Command list appending can occur independently of command queue submission.
- Command list submission can occur to more than one command queue.

The following diagram illustrates the hierarchy of command lists and command queues to the device:



2.3.1 Command Queues

A command queue represents a logical input stream to the device, tied to a physical input stream via an ordinal at creation time.

Creation

- The application may explicitly bind the command queue to a physical input stream, or allow the driver to choose dynamically, based on usage.
- Multiple command queues may be created that use the same physical input stream. For example, an application may create a command queue per Host thread with different scheduling priorities.
- However, because each command queue allocates a logical hardware context, an application should avoid creating multiple command queues for the same physical input stream with the same priority due to possible performance penalties with hardware context switching.
- The maximum number of command queues an application can create is limited by device-specific resources; e.g., the maximum number of logical hardware contexts supported by the device. This can be queried from ::ze_device_properties_t.maxCommandQueues.
- The maximum number of simultaneous compute command queues per device is queried from ::ze_device_properties_t.numAsyncComputeEngines.
- The maximum number of simultaneous copy command queues per device is queried from ::ze_device_properties_t.numAsyncCopyEngines.
- All command lists executed on a command queue are guaranteed to only execute on its single, physical input stream; e.g., copy commands in a compute command list / queue will execute via the compute engine, not the copy engine.

The following pseudo-code demonstrates a basic sequence for creation of command queues:

```
// Create a command queue
ze_command_queue_desc_t commandQueueDesc = {
    ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT,
    ZE_COMMAND_QUEUE_FLAG_NONE,
    ZE_COMMAND_QUEUE_MODE_DEFAULT,
    ZE_COMMAND_QUEUE_PRIORITY_NORMAL,
    0
};
ze_command_queue_handle_t hCommandQueue;
zeCommandQueueCreate(hDevice, &commandQueueDesc, &hCommandQueue);
...
```

Execution

- Command lists submitted to a command queue are **immediately** executed in a fifo manner.
- Command queue submission is free-treaded, allowing multiple Host threads to share the same command queue.
- If multiple Host threads enter the same command queue simultaneously, then execution order is undefined.
- Command lists created with ::ZE_COMMAND_LIST_FLAG_COPY_ONLY may only be submitted to command queues created with ::ZE_COMMAND_QUEUE_FLAG_COPY_ONLY.

Destruction

- The application is responsible for making sure the device is not currently executing from a command queue before it is deleted. This is typically done by tracking command queue fences, but may also be handled by calling ::zeCommandQueueSynchronize.

2.3.2 Command Lists

A command list represents a sequence of commands for execution on a command queue.

Creation

- A command list is created for a device to allow device-specific appending of commands.
- A command list can be copied to create another command list. The application may use this to copy a command list for use on a different device.

Appending

- There is no implicit binding of command lists to Host threads. Therefore, an application may share a command list handle across multiple Host threads. However, the application is responsible for ensuring that multiple Host threads do not access the same command list simultaneously.
- By default, commands are executed in the same order in which they are appended. However, an application may allow the driver to optimize the ordering by using ::ZE_COMMAND_LIST_FLAG_RELAXED_ORDERING. Reordering is guaranteed to be only occur between barriers and synchronization primitives.
- By default, commands submitted to a command list are optimized for execution by balancing both device throughput and Host latency. For very low-level latency usage-models, applications should use immediate command lists. For usage-models where maximum throughput is desired, applications should use ::ZE_COMMAND_LIST_FLAG_MAXIMIZE_THROUGHPUT.
- By default, commands submitted to a command list may be optimized by the driver to fully exploit the concurrency of the device by distributing commands across multiple engines and/or sub-devices. If the application prefers to opt-out of these optimizations, such as when the application plans to perform this distribution itself, then it should use ::ZE_COMMAND_LIST_FLAG_EXPLICIT_ONLY.

The following pseudo-code demonstrates a basic sequence for creation of command lists:

```
// Create a command list
ze_command_list_desc_t commandListDesc = {
    ZE_COMMAND_LIST_DESC_VERSION_CURRENT,
    ZE_COMMAND_LIST_FLAG_NONE
};
ze_command_list_handle_t hCommandList;
zeCommandListCreate(hDevice, &commandListDesc, &hCommandList);
...
```

Submission

- There is no implicit association between a command list and a command queue. Therefore, a command list may be submitted to any, or multiple command queues. However, if a command list is meant to be submitted to a copy-only command queue then the ::ZE_COMMAND_LIST_FLAG_COPY_ONLY must be set at creation.
- The application is responsible for calling close before submission to a command queue.
- Command lists do not inherit state from other command lists executed on the same command queue. i.e. each command list begins execution in its own default state.
- A command list may be submitted multiple times. It is up to the application to ensure that the command list can be executed multiple times. Events, for example, must be explicitly reset prior to re-execution.

The following pseudo-code demonstrates submission of commands to a command queue, via a command list:

```
...
// finished appending commands (typically done on another thread)
zeCommandListClose(hCommandList);

// Execute command list in command queue
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);

// synchronize host and device
zeCommandQueueSynchronize(hCommandQueue, UINT32_MAX);

// Reset (recycle) command list for new commands
zeCommandListReset(hCommandList);
...
```

Recycling

- A command list may be recycled to avoid the overhead of frequent creation and destruction.
- The application is responsible for making sure the device is not currently executing from a command list before it is reset. This should be handled by tracking a completion event associated with the command list.
- The application is responsible for making sure the device is not currently executing from a command list before it is deleted. This should be handled by tracking a completion event associated with the command list.

Low-Latency Immediate Command Lists

A special type of command list can be used for very low-latency submission usage-models.

- An immediate command list is both a command list and an implicit command queue.
- An immediate command list is created using a command queue descriptor.
- Commands submitted to an immediate command list are immediately executed on the device.
- An immediate command list is not required to be closed or reset. However, usage will be honored, and expected behaviors will be followed.

The following pseudo-code demonstrates a basic sequence for creation and usage of immediate command lists:

```
// Create an immediate command list
ze_command_queue_desc_t commandQueueDesc = {
    ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT,
```

(continues on next page)

(continued from previous page)

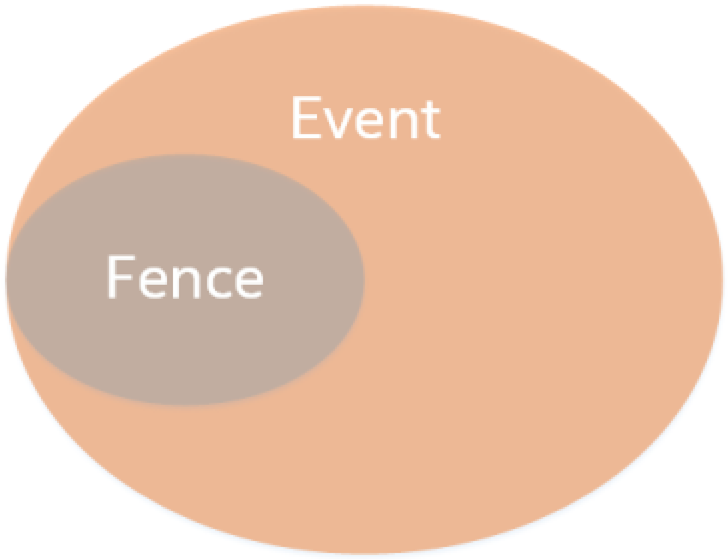
```
    ZE_COMMAND_QUEUE_FLAG_NONE,
    ZE_COMMAND_QUEUE_MODE_DEFAULT,
    ZE_COMMAND_QUEUE_PRIORITY_NORMAL,
    0
};
ze_command_list_handle_t hCommandList;
zeCommandListCreateImmediate(hDevice, &commandQueueDesc, &hCommandList);

// Immediately submit a kernel to the device
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);
...
```

2.4 Synchronization Primitives

There are two types of synchronization primitives:

1. *Fences* - used to communicate to the host that command queue execution has completed.
2. *Events* - used as fine-grain host-to-device, device-to-host or device-to-device execution and memory dependencies.

The following diagram illustrates the relationship of capabilities of these types of synchronization primitives:



The following are the motivations for separating the different types of synchronization primitives:

- Allows device-specific optimizations for certain types of primitives:
  - fences may share device memory with all other fences within the same command queue.
  - events may be implemented using pipelined operations as part of the program execution.
  - fences are implicit, coarse-grain execution and memory barriers.
  - events optionally cause fine-grain execution and memory barriers.
- Allows distinction on which type of primitive may be shared across devices.

Generally. Events are generic synchronization primitives that can be used across many different usage-models, including those of fences. However, this generality comes with some cost in memory overhead and efficiency.

2.4.1 Fences

A fence is a heavyweight synchronization primitive used to communicate to the host that command list execution within a command queue has completed.

- A fence is associated with a single command queue.
- A fence can only be signaled from a device's command queue (e.g. between execution of command lists) and can only be waited upon from the host.
- A fence guarantees both execution completion and memory coherency, across the device and host, prior to being signaled.
- A fence only has two states: not signaled and signaled.
- A fence can only be reset from the Host.
- A fence cannot be shared across processes.

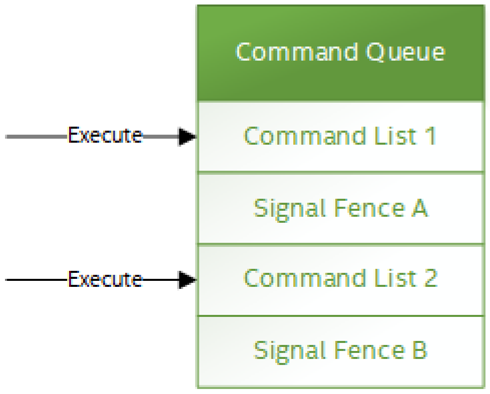
The following pseudo-code demonstrates a sequence for creation, submission and querying of a fence:

```
// Create fence
ze_fence_desc_t fenceDesc = {
    ZE_FENCE_DESC_VERSION_CURRENT,
    ZE_FENCE_FLAG_NONE
};
ze_fence_handle_t hFence;
zeFenceCreate(hCommandQueue, &fenceDesc, &hFence);

// Execute a command list with a signal of the fence
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, hFence);

// Wait for fence to be signaled
zeFenceHostSynchronize(hFence, UINT32_MAX);
zeFenceReset(hFence);
...
```

The primary usage model(s) for fences are to notify the Host when a command list has finished execution to allow:

- Recycling of memory and images
- Recycling of command lists
- Recycling of other synchronization primitives
- Explicit memory residency.

The following diagram illustrates fences signaled after command lists on execution:



2.4.2 Events

An event is used to communicate fine-grain host-to-device, device-to-host or device-to-device dependencies from within a command list.

- An event can be:
  - signaled from within a device's command list and waited upon within the same command list
  - signaled from within a device's command list and waited upon from the host, another command queue or another device
  - signaled from the host, and waited upon from within a device's command list.
- An event only has two states: not signaled and signaled.
- An event doesn't implicitly reset. Signaling a signaled event (or resetting an unsignaled event) is valid and has no effect on the state of the event.
- An event can be explicitly reset from the Host or device.
- An event can be appended into multiple command lists simultaneously.
- An event can be shared across devices and processes.
- An event can invoke an execution and/or memory barrier; which should be used sparingly to avoid device underutilization.
- There are no protections against events causing deadlocks, such as circular waits scenarios.
  - These problems are left to the application to avoid.
- An event intended to be signaled by the host, another command queue or another device after command list submission to a command queue may prevent subsequent forward progress within the command queue itself.
  - This can create bubbles in the pipeline or deadlock situations if not correctly scheduled.

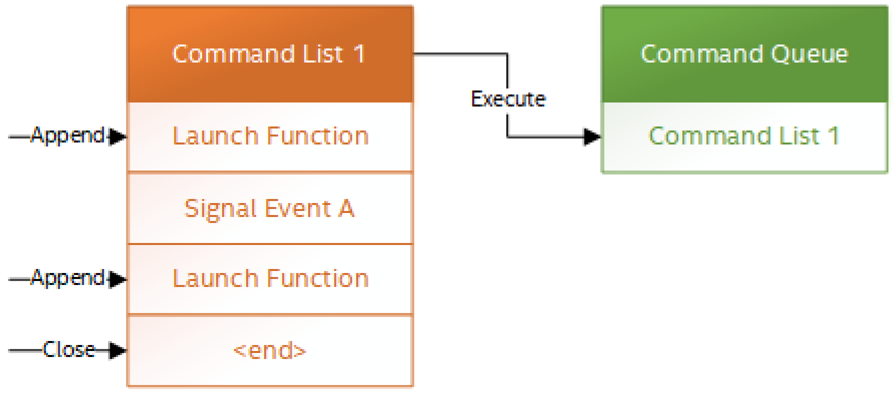
An event pool is used for creation of individual events:

- An event pool reduces the cost of creating multiple events by allowing underlying device allocations to be shared by events with the same properties
- An event pool can be shared via IPC; allowing sharing blocks of events rather than sharing each individual event

The following pseudo-code demonstrates a sequence for creation and submission of an event:

```
// Create event pool
ze_event_pool_desc_t eventPoolDesc = {
    ZE_EVENT_POOL_DESC_VERSION_CURRENT,
    ZE_EVENT_POOL_FLAG_HOST_VISIBLE, // all events in pool are visible to Host
    1
};
ze_event_pool_handle_t hEventPool;
zeEventPoolCreate(hDriver, &eventPoolDesc, 0, nullptr, &hEventPool);

ze_event_desc_t eventDesc = {
    ZE_EVENT_DESC_VERSION_CURRENT,
    0,
    ZE_EVENT_SCOPE_FLAG_NONE,
    ZE_EVENT_SCOPE_FLAG_HOST  // ensure memory coherency across device and Host after␣
↪event completes
};
ze_event_handle_t hEvent;
zeEventCreate(hEventPool, &eventDesc, &hEvent);

// Append a signal of an event into the command list after the kernel executes
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent, 0,␣
↪nullptr);

// Execute the command list with the signal
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);
...
```

The following diagram illustrates an event being signaled between kernels within a command list:



2.5 Barriers

There are two types of barriers:

1. **Execution Barriers** - used to communicate execution dependencies between commands within a command list or across command queues, devices and/or Host.
2. **Memory Barriers** - used to communicate memory coherency dependencies between commands within a command list or across command queues, devices and/or Host.

The following pseudo-code demonstrates a sequence for submission of a brute-force execution and global memory barrier:

```
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);

// Append a barrier into a command list to ensure hKernel1 completes before hKernel2␣
↪begins
zeCommandListAppendBarrier(hCommandList, nullptr, 0, nullptr);

zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);
...
```

2.5.1 Execution Barriers

Commands executed on a command list are only guaranteed to start in the same order in which they are submitted; i.e.?there is no implicit definition of the order of completion.

- Fences provide implicit, coarse-grain control to indicate that all previous commands must complete prior to the fence being signaled.
- Events provide explicit, fine-grain control over execution dependencies between commands; allowing more opportunities for concurrent execution and higher device utilization.

The following pseudo-code demonstrates a sequence for submission of a fine-grain execution-only dependency using events:

```
ze_event_desc_t event1Desc = {
    ZE_EVENT_DESC_VERSION_CURRENT,
    0,
    ZE_EVENT_SCOPE_FLAG_NONE, // no memory/cache coherency required on signal
    ZE_EVENT_SCOPE_FLAG_NONE  // no memory/cache coherency required on wait
};
ze_event_handle_t hEvent1;
zeEventCreate(hEventPool, &event1Desc, &hEvent1);

// Ensure hKernel1 completes before signaling hEvent1
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent1, 0,␣
↪nullptr);

// Ensure hEvent1 is signaled before starting hKernel2
zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 1, &
↪hEvent1);
...
```

2.5.2 Memory Barriers

Commands executed on a command list are *not* guaranteed to maintain memory coherency with other commands; i.e. there is no implicit memory or cache coherency.

- Fences provide implicit, coarse-grain control to indicate that all caches and memory are coherent across the device and Host prior to the fence being signaled.
- Events provide explicit, fine-grain control over cache and memory coherency dependencies between commands; allowing more opportunities for concurrent execution and higher device utilization.

The following pseudo-code demonstrates a sequence for submission of a fine-grain memory dependency using events:

```
ze_event_desc_t event1Desc = {
    ZE_EVENT_DESC_VERSION_CURRENT,
    0,
    ZE_EVENT_SCOPE_FLAG_DEVICE, // ensure memory coherency across device before event␣
↪signaled
    ZE_EVENT_SCOPE_FLAG_NONE
};
ze_event_handle_t hEvent1;
zeEventCreate(hEventPool, &event1Desc, &hEvent1);

// Ensure hKernel1 memory writes are fully coherent across the device before␣
↪signaling hEvent1
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, hEvent1, 0,␣
↪nullptr);

// Ensure hEvent1 is signaled before starting hKernel2
zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 1, &
↪hEvent1);
...
```

2.5.3 Range-based Memory Barriers

Range-based memory barriers provide explicit control of which cachelines require coherency.

The following pseudo-code demonstrates a sequence for submission of a range-based memory barrier:

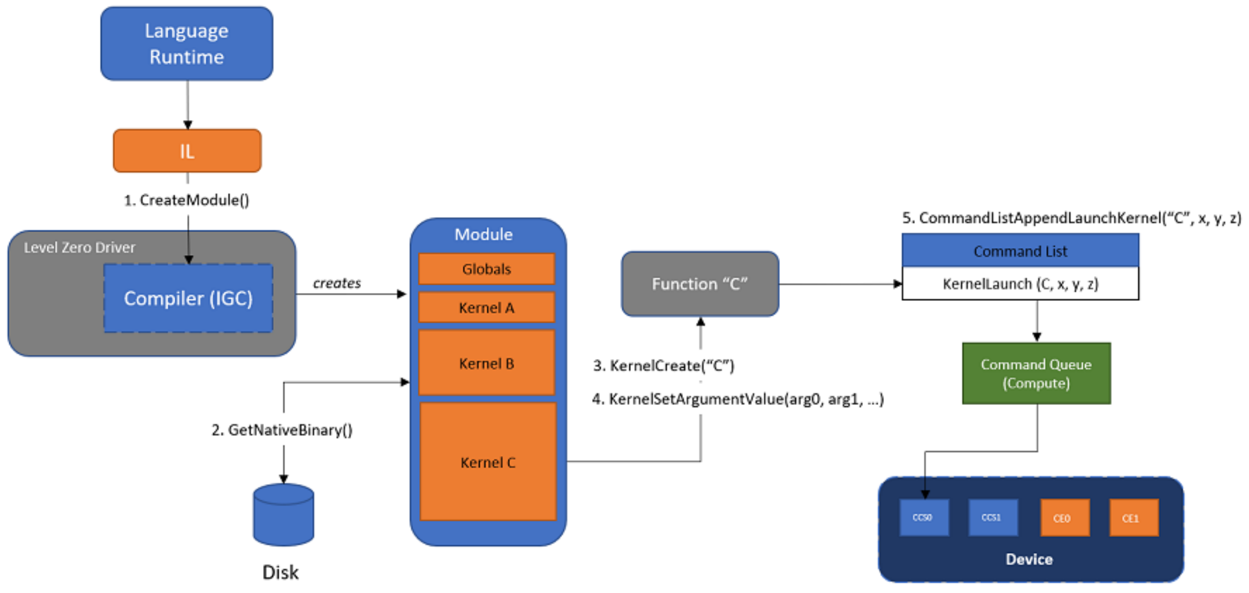
```
zeCommandListAppendLaunchKernel(hCommandList, hKernel1, &launchArgs, nullptr, 0,␣
↪nullptr);

// Ensure memory range is fully coherent across the device after hKernel1 and before␣
↪hKernel2
zeCommandListAppendMemoryRangesBarrier(hCommandList, 1, &size, &ptr, nullptr, 0,␣
↪nullptr);

zeCommandListAppendLaunchKernel(hCommandList, hKernel2, &launchArgs, nullptr, 0,␣
↪nullptr);
...
```

2.6 Modules and Kernels

There are multiple levels of constructs needed for executing kernels on the device:

1. *Modules* represent a single translation unit that consists of kernels that have been compiled together.
2. *Kernels* represent the kernel within the module that will be launched directly from a command list.

The following diagram provides a high-level overview of the major parts of the system.



2.6.1 Modules

Modules can be created from an IL or directly from native format using ::zeModuleCreate.

- ::zeModuleCreate takes a format argument that specifies the input format.
- ::zeModuleCreate performs a compilation step when format is IL.

The following pseudo-code demonstrates a sequence for creating a module from an OpenCL kernel:

```
__kernel void image_scaling( __read_only  image2d_t src_img,
                             __write_only image2d_t dest_img,
                                          uint WIDTH,     // resized width
                                          uint HEIGHT )   // resized height
{
    int2       coor = (int2)( get_global_id(0), get_global_id(1) );
    float2 normCoor = convert_float2(coor) / (float2)( WIDTH, HEIGHT );

    float4     color = read_imagef( src_img, SMPL_PREF, normCoor );

    write_imagef( dest_img, coor, color );
}
...
```

```
// OpenCL C kernel has been compiled to SPIRV IL (pImageScalingIL)
ze_module_desc_t moduleDesc = {
    ZE_MODULE_DESC_VERSION_CURRENT,
    ZE_MODULE_FORMAT_IL_SPIRV,
    ilSize,
    pImageScalingIL,
    nullptr,
    nullptr
};
ze_module_handle_t hModule;
zeModuleCreate(hDevice, &moduleDesc, &hModule, nullptr);
...
```

Module Build Options

Module build options can be passed with ::ze_module_desc_t as a string.

Build Option	Description	Default	Device Support
-ze-opt-disable	Disable optimizations.	Disabled	All
-ze-opt-greater-than-4GB-buffer-required	Use 64-bit offset calculations for buffers.	Disabled	GPU
-ze-opt-large-register-file	Increase number of registers available to threads.	Disabled	GPU

Module Specialization Constants

SPIR-V supports specialization constants that allow certain constants to be updated to new values during runtime execution. Each specialization constant in SPIR-V has an identifier and default value. The ::zeModuleCreate function allows for an array of constants and their corresponding identifiers to be passed in to override the constants in the SPIR-V module.

```
// Spec constant overrides for group size.
ze_module_constants_t specConstants = {
    3,
    pGroupSizeIds,
    pGroupSizeValues
};
// OpenCL C kernel has been compiled to SPIRV IL (pImageScalingIL)
ze_module_desc_t moduleDesc = {
    ZE_MODULE_DESC_VERSION_CURRENT,
    ZE_MODULE_FORMAT_IL_SPIRV,
    ilSize,
    pImageScalingIL,
    nullptr,
    &specConstants
};
ze_module_handle_t hModule;
zeModuleCreate(hDevice, &moduleDesc, &hModule, nullptr);
...
```

Note: Specialization constants are only handled at module create time and therefore if you need to change them then you'll need to compile a new module.

Module Build Log

The ::zeModuleCreate function can optionally generate a build log object ::ze_module_build_log_handle_t.

```
...
ze_module_build_log_handle_t buildlog;
ze_result_t result = zeModuleCreate(hDevice, &desc, &module, &buildlog);

// Only save build logs for module creation errors.
if (result != ZE_RESULT_SUCCESS)
{
    size_t szLog = 0;
```

(continues on next page)

(continued from previous page)

```
    zeModuleBuildLogGetString(buildlog, &szLog, nullptr);

    char_t* strLog = allocate(szLog);
    zeModuleBuildLogGetString(buildlog, &szLog, strLog);

    // Save log to disk.
    ...

    free(strLog);
}

zeModuleBuildLogDestroy(buildlog);
```

Module Caching with Native Binaries

Disk caching of modules is not supported by the driver. If a disk cache for modules is desired, then it is the responsibility of the application to implement this using ::zeModuleGetNativeBinary.

```
...
// compute hash for pIL and check cache.
...

if (cacheUpdateNeeded)
{
    size_t szBinary = 0;
    zeModuleGetNativeBinary(hModule, &szBinary, nullptr);

    uint8_t* pBinary = allocate(szBinary);
    zeModuleGetNativeBinary(hModule, &szBinary, pBinary);

    // cache pBinary for corresponding IL
    ...

    free(pBinary);
}
```

Also, note that the native binary will retain all debug information that is associated with the module. This allows debug capabilities for modules that are created from native binaries.

Built-in Kernels

Built-in kernels are not supported but can be implemented by an upper level runtime or library using the native binary interface.

2.6.2 Kernels

A Kernel is a reference to a kernel within a module. The Kernel object supports both explicit and implicit kernel arguments along with data needed for launch.

The following pseudo-code demonstrates a sequence for creating a kernel from a module:

```
ze_kernel_desc_t kernelDesc = {
    ZE_KERNEL_DESC_VERSION_CURRENT,
    ZE_KERNEL_FLAG_NONE,
    "image_scaling"
};
ze_kernel_handle_t hKernel;
zeKernelCreate(hModule, &kernelDesc, &hKernel);
...
```

Kernel Attributes and Properties

Use ::zeKernelSetAttribute to set attributes for a kernel object.

```
// Kernel performs indirect device access.
bool_t isIndirect = true;
zeKernelSetAttribute(hKernel, ZE_KERNEL_ATTR_INDIRECT_DEVICE_ACCESS, sizeof(bool_t), &
↪isIndirect);
...
```

Use ::zeKernelSetAttribute to get attributes for a kernel object.

```
// Does kernel perform indirect device access.
zeKernelGetAttribute(hKernel, ZE_KERNEL_ATTR_INDIRECT_DEVICE_ACCESS, sizeof(bool_t), &
↪isIndirect);
...

uint32_t strSize = 0; // Size of string + null terminator
zeKernelGetAttribute(hKernel, ZE_KERNEL_ATTR_SOURCE_ATTRIBUTE, &strSize, nullptr );
char* pAttributes = allocate(strSize);
zeKernelGetAttribute(hKernel, ZE_KERNEL_ATTR_SOURCE_ATTRIBUTE, &strSize, pAttributes␣
↪);
...
```

See ::ze_kernel_attribute_t for more information on the “set” and “get” attributes.

Use ::zeKernelGetProperties to query invariant properties from a kernel object.

```
...
ze_kernel_properties_t kernelProperties;

//
zeKernelGetProperties(hKernel, &kernelProperties);
...
```

See ::ze_kernel_properties_t for more information for kernel properties.

2.6.3 Execution

Kernel Group Size

The group size for a kernel can be set using ::zeKernelSetGroupSize. If a group size is not set prior to appending a kernel into a command list then a default will be chosen. The group size can be updated over a series of append operations. The driver will copy the group size information when appending the kernel into the command list.

```
zeKernelSetGroupSize(hKernel, groupSizeX, groupSizeY, 1);

...
```

The API supports a query for suggested group size when providing the global size. This function ignores the group size that was set on the kernel using ::zeKernelSetGroupSize.

```
// Find suggested group size for processing image.
uint32_t groupSizeX;
uint32_t groupSizeY;
zeKernelSuggestGroupSize(hKernel, imageWidth, imageHeight, 1, &groupSizeX, &
↪groupSizeY, nullptr);

zeKernelSetGroupSize(hKernel, groupSizeX, groupSizeY, 1);

...
```

Kernel Arguments

Kernel arguments represent only the explicit kernel arguments that are within ?brackets? e.g.?func(arg1, arg2, ?).

- Use ::zeKernelSetArgumentValue to setup arguments for a kernel launch.
- The AppendLaunchKernel command will make a copy the kernel arguments to send to the device.
- Kernel arguments can be updated at any time and used across multiple append calls.

The following pseudo-code demonstrates a sequence for setting kernel args and launching the kernel:

```
// Bind arguments
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_image_handle_t), &src_image);
zeKernelSetArgumentValue(hKernel, 1, sizeof(ze_image_handle_t), &dest_image);
zeKernelSetArgumentValue(hKernel, 2, sizeof(uint32_t), &width);
zeKernelSetArgumentValue(hKernel, 3, sizeof(uint32_t), &height);

ze_group_count_t launchArgs = { numGroupsX, numGroupsY, 1 };

// Append launch kernel
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);

// Update image pointers to copy and scale next image.
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_image_handle_t), &src2_image);
zeKernelSetArgumentValue(hKernel, 1, sizeof(ze_image_handle_t), &dest2_image);

// Append launch kernel
```

(continues on next page)

(continued from previous page)

```
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);

...
```

Kernel Launch

In order to launch a kernel on the device an application must call one of the CommandListAppendLaunch* functions for a command list. The most basic version of these is ::zeCommandListAppendLaunchKernel which takes a command list, kernel handle, launch arguments, and an optional synchronization event used to signal completion. The launch arguments contain thread group dimensions.

```
// compute number of groups to launch based on image size and group size.
uint32_t numGroupsX = imageWidth / groupSizeX;
uint32_t numGroupsY = imageHeight / groupSizeY;

ze_group_count_t launchArgs = { numGroupsX, numGroupsY, 1 };

// Append launch kernel
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);
```

The function ::zeCommandListAppendLaunchKernelIndirect allows the launch parameters to be supplied indirectly in a buffer that the device reads instead of the command itself. This allows for the previous operations on the device to generate the parameters.

```
ze_group_count_t* pIndirectArgs;

...
zeDriverAllocDeviceMem(hDriver, &desc, sizeof(ze_group_count_t), sizeof(uint32_t),␣
↪hDevice, &pIndirectArgs);

// Append launch kernel - indirect
zeCommandListAppendLaunchKernelIndirect(hCommandList, hKernel, &pIndirectArgs,␣
↪nullptr, 0, nullptr);
```

Cooperative Kernels

Cooperative kernels allow sharing of data and synchronization across all launched groups in a safe manner. To support this there is a ::zeCommandListAppendLaunchCooperativeKernel that allows launching groups that can cooperate with each other. Finally, there is a ::zeKernelSuggestMaxCooperativeGroupCount function that suggests a maximum group count size that the device supports.

2.6.4 Sampler

The API supports Sampler objects that represent state needed for sampling images from within kernels. The ::zeSamplerCreate function takes a sampler descriptor (::ze_sampler_desc_t):

Sampler Field	Description
Address Mode	Determines how out-of-bounds accessse are handled. See ::ze_sampler_address_mode_t.
Filter Mode	Specifies which filtering mode to use. See ::ze_sampler_filter_mode_t
Normalized	Specifies whether coordinates for addressing image are normalized [0,1] or not.

The following is sample for code creating a sampler object and passing it as a kernel argument:

```
// Setup sampler for linear filtering and clamp out of bounds accesses to edge.
ze_sampler_desc_t desc = {
    ZE_SAMPLER_DESC_VERSION_CURRENT,
    ZE_SAMPLER_ADDRESS_MODE_CLAMP,
    ZE_SAMPLER_FILTER_MODE_LINEAR,
    false
    };
ze_sampler_handle_t sampler;
zeSamplerCreate(hDevice, &desc, &sampler);
...

// The sampler can be passed as a kernel argument.
zeKernelSetArgumentValue(hKernel, 0, sizeof(ze_sampler_handle_t), &sampler);

// Append launch kernel
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);
```

2.7 Advanced

2.7.1 Sub-Device Support

The API allows support for sub-devices which can enable finer grained control of scheduling and memory allocation to a sub-partition of the device. There are functions to query and obtain a sub-device but outside of these functions there are no distinction between sub-devices and devices.

Use ::zeDeviceGetSubDevices to confirm sub-devices are supported and to obtain a sub-device handle. There are additional device properties in ::ze_device_properties_t for sub-devices to confirm a device is a sub-device and to query the sub-device id. This is useful when needing to pass a sub-device handle to another library.

To allocate memory and dispatch tasks to a specific sub-device then obtain the sub-device handle and use this with memory and command queue/lists APIs. Local memory allocation will be placed in the local memory that is attached to the sub-device. An out-of-memory error indicates that there is not enough local sub-device memory for the allocation. The driver will not try and spill sub-device allocations over to another sub-device's local memory. However, the application can retry using the parent device and the driver will decide where to place the allocation.

One thing to note is that the ordinal that is used when creating a command queue is relative to the sub-device. This ordinal specifies which physical compute queue on the device or sub-device to map the logical queue to. The application needs to query ::ze_device_properties_t.numAsyncComputeEngines from the sub-device to determine how to set this ordinal. See ::ze_command_queue_desc_t for more details.

A 16-byte unique device identifier (uuid) can be obtained for a device or sub-device using ::zeDeviceGetProperties.

```
// Query for all sub-devices of the device
uint32_t subdeviceCount = 0;
zeDeviceGetSubDevices(hDevice, &subdeviceCount, nullptr);

ze_device_handle_t* allSubDevices = allocate(subdeviceCount * sizeof(ze_device_handle_
↪t));
zeDeviceGetSubDevices(hDevice, &subdeviceCount, &allSubDevices);

// Desire is to allocate and dispatch work to sub-device 2.
assert(subdeviceCount >= 3);
ze_device_handle_t hSubdevice = allSubDevices[2];

// Query sub-device properties.
ze_device_properties_t subdeviceProps;
zeDeviceGetProperties(hSubdevice, &subdeviceProps);

assert(subdeviceProps.isSubdevice == true); // Ensure that we have a handle to a sub-
↪device.
assert(subdeviceProps.subdeviceId == 2);    // Ensure that we have a handle to the␣
↪sub-device we asked for.

void* pMemForSubDevice2;
zeDriverAllocDeviceMem(hDriver, &desc, memSize, sizeof(uint32_t), hSubdevice, &
↪pMemForSubDevice2);
...

...
// Check that cmd queue ordinal that was chosen is valid.
assert(desc.ordinal < subdeviceProps.numAsyncComputeEngines);

ze_command_queue_handle_t commandQueueForSubDevice2;
zeCommandQueueCreate(hSubdevice, &desc, &commandQueueForSubDevice2);
...
```

2.7.2 Device Residency

For devices that do not support page-faults, the driver must ensure that all pages that will be accessed by the kernel are resident before program execution. This can be determined by checking ::ze_device_properties_t.onDemandPageFaultsSupported.

In most cases, the driver implicitly handles residency of allocations for device access. This can be done by inspecting API parameters, including kernel arguments. However, in cases where the devices does **not** support page-faulting *and* the driver is incapable of determining whether an allocation will be accessed by the device, such as multiple levels of indirection, there are two methods available:

1. The application may set the ::ZE_KERNEL_FLAG_FORCE_RESIDENCY flag during program creation to force all device allocations to be resident during execution.

   - in addition, the application should indicate the type of allocations that will be indirectly accessed using ::ze_kernel_attribute_t (ZE_KERNEL_ATTR_INDIRECT_HOST_ACCESS, DEVICE_ACCESS, or SHARED_ACCESS).
   - if the driver is unable to make all allocations resident, then the call to ::zeCommandQueueExecuteCommandLists will return ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY

2. Explcit ::zeDeviceMakeMemoryResident APIs are included for the application to dynamically change residency as needed. (Windows-only)

- if the application over-commits device memory, then a call to ::zeDeviceMakeMemoryResident will return ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY

If the application does not properly manage residency for these cases then the device may experience unrecoverable page-faults.

The following pseudo-code demonstrate a sequence for using coarse-grain residency control for indirect arguments:

```
struct node {
    node* next;
};
node* begin = nullptr;
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin);
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin->next);
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin->next->next);

// 'begin' is passed as kernel argument and appended into command list
bool hasIndirectHostAccess = true;
zeKernelSetAttribute(hFuncArgs, ZE_KERNEL_ATTR_INDIRECT_HOST_ACCESS, sizeof(bool), &
↪hasIndirectHostAccess);
zeKernelSetArgumentValue(hKernel, 0, sizeof(node*), &begin);
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);

...

zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, nullptr);
...
```

The following pseudo-code demonstrate a sequence for using fine-grain residency control for indirect arguments:

```
struct node {
    node* next;
};
node* begin = nullptr;
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin);
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin->next);
zeDriverAllocHostMem(hDriver, &desc, sizeof(node), 1, &begin->next->next);

// 'begin' is passed as kernel argument and appended into command list
zeKernelSetArgumentValue(hKernel, 0, sizeof(node*), &begin);
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &launchArgs, nullptr, 0,␣
↪nullptr);
...

// Make indirect allocations resident before enqueuing
zeDeviceMakeMemoryResident(hDevice, begin->next, sizeof(node));
zeDeviceMakeMemoryResident(hDevice, begin->next->next, sizeof(node));

zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList, hFence);

// wait until complete
zeFenceHostSynchronize(hFence, UINT32_MAX);

// Finally, evict to free device resources
zeDeviceEvictMemory(hDevice, begin->next, sizeof(node));
zeDeviceEvictMemory(hDevice, begin->next->next, sizeof(node));
...
```

2.7.3 OpenCL Interoperability

Interoperability with OpenCL is currently only supported *from* OpenCL *to* Level-Zero for a subset of types. The APIs are designed to be OS agnostics and allow implementations to optimize for unified device drivers; while allowing less optimal interoperability across different device types and/or vendors.

There are three OpenCL types that can be shared for interoperability:

1. **cl_mem** - an OpenCL buffer object
2. **cl_program** - an OpenCL program object
3. **cl_command_queue** - an OpenCL command queue object

cl_mem

OpenCL buffer objects may be registered for use as a Level-Zero device memory allocation. Registering an OpenCL buffer object with Level-Zero merely obtains a pointer to the underlying device memory allocation and does not alter the lifetime of the device memory underlying the OpenCL buffer object. Freeing the Level-Zero device memory allocation effectively "un-registers" the allocation from Level-Zero, and should be performed before the OpenCL buffer object is destroyed. Using the Level-Zero device memory allocation after destroying its associated OpenCL buffer object will result in undefined behavior.

Applications are responsible for enforcing memory consistency for shared buffer objects using existing OpenCL and/or Level-Zero APIs.

cl_program

Level-Zero modules are always in a compiled state and therefore prior to retrieving an ::ze_module_handle_t from a cl_program the caller must ensure the cl_program is compiled and linked.

cl_command_queue

Sharing OpenCL command queues provide opportunities to minimize transition costs when submitting work from an OpenCL queue followed by submitting work to Level-Zero command queue and vice-versa. Enqueuing Level-Zero command lists to Level-Zero command queues are immediately submitted to the device. OpenCL implementations, however, may not necessarily submit tasks to the device unless forced by explicit OpenCL API such as clFlush or clFinish. To minimize overhead between sharing command queues, applications must explicitly submit OpenCL command queues using clFlush, clFinish or similar operations prior to enqueuing a Level-Zero command list. Failing to explicitly submit device work may result in undefined behavior.

Sharing an OpenCL command queue doesn't alter the lifetime of the API object. It provides knowledge for the driver to potentially reuse some internal resources which may have noticeable overhead when switching the resources.

Memory contents as reflected by any caching schemes will be consistent such that, for example, a memory write in an OpenCL command queue can be read by a subsequent Level-Zero command list without any special application action. The cost to ensure memory consistency may be implementation dependent. The performance of sharing command queues will be no worse than an application submitting work to OpenCL, calling clFinish followed by submitting a Level-Zero command list. In most cases, command queue sharing may be much more efficient.

2.7.4 Inter-Process Communication

There are two types of Inter-Process Communication (IPC) APIs for using Level-Zero allocations across processes:

1. Memory
2. Events

Memory

The following code examples demonstrate how to use the memory IPC APIs:

1. First, the allocation is made, packaged, and sent on the sending process:

```
void* dptr = nullptr;
zeDriverAllocDeviceMem(hDriver, &desc, size, alignment, hDevice, &dptr);

ze_ipc_mem_handle_t hIPC;
zeDriverGetMemIpcHandle(hDriver, dptr, &hIPC);

// Method of sending to receiving process is not defined by Level-Zero:
send_to_receiving_process(hIPC);
```

2. Next, the allocation is received and un-packaged on the receiving process:

```
// Method of receiving from sending process is not defined by Level-Zero:
ze_ipc_mem_handle_t hIPC;
hIPC = receive_from_sending_process();

void* dptr = nullptr;
zeDriverOpenMemIpcHandle(hDriver, hDevice, hIPC, ZE_IPC_MEMORY_FLAG_NONE, &dptr);
```

3. Each process may now refer to the same device memory allocation via its `dptr`. Note, there is no guaranteed address equivalence for the values of `dptr` in each process.
4. To cleanup, first close the handle in the receiving process:

```
zeDriverCloseMemIpcHandle(hDriver, dptr);
```

5. Finally, free the device pointer in the sending process:

```
zeDriverFreeMem(hDriver, dptr);
```

Events

The following code examples demonstrate how to use the event IPC APIs:

1. First, the event pool is created, packaged, and sent on the sending process:

```
// create event pool
ze_event_pool_desc_t eventPoolDesc = {
    ZE_EVENT_POOL_DESC_VERSION_CURRENT,
    ZE_EVENT_POOL_FLAG_IPC | ZE_EVENT_POOL_FLAG_HOST_VISIBLE,
    10
};
ze_event_pool_handle_t hEventPool;
zeEventPoolCreate(hDriver, &eventPoolDesc, 1, &hDevice, &hEventPool);
```

(continues on next page)

(continued from previous page)

```

// get IPC handle and send to another process
ze_ipc_event_pool_handle_t hIpcEvent;
zeEventPoolGetIpcHandle(hEventPool, &hIpcEventPool);
send_to_receiving_process(hIpcEventPool);
```

2. Next, the event pool is received and un-packaged on the receiving process:

```
// get IPC handle from other process
ze_ipc_event_pool_handle_t hIpcEventPool;
receive_from_sending_process(&hIpcEventPool);

// open event pool
ze_event_pool_handle_t hEventPool;
zeEventPoolOpenIpcHandle(hDriver, hIpcEventPool, &hEventPool);
```

3. Each process may now refer to the same device event allocation via its handle.

   a. receiving process creates event at location

```
ze_event_handle_t hEvent;
ze_event_desc_t eventDesc = {
    ZE_EVENT_DESC_VERSION_CURRENT,
    5,
    ZE_EVENT_SCOPE_FLAG_NONE,
    ZE_EVENT_SCOPE_FLAG_HOST, // ensure memory coherency across device␣
↪and Host after event signaled
};
zeEventCreate(hEventPool, &eventDesc, &hEvent);

// submit kernel and signal event when complete
zeCommandListAppendLaunchKernel(hCommandList, hKernel, &args, hEvent, 0,␣
↪nullptr);
zeCommandListClose(hCommandList);
zeCommandQueueExecuteCommandLists(hCommandQueue, 1, &hCommandList,␣
↪nullptr);
```

   b. sending process creates event at same location

```
ze_event_handle_t hEvent;
ze_event_desc_t eventDesc = {
    ZE_EVENT_DESC_VERSION_CURRENT,
    5,
    ZE_EVENT_SCOPE_FLAG_NONE,
    ZE_EVENT_SCOPE_FLAG_HOST, // ensure memory coherency across device␣
↪and Host after event signaled
};
zeEventCreate(hEventPool, &eventDesc, &hEvent);

zeEventHostSynchronize(hEvent, UINT32_MAX);
```

   Note, there is no guaranteed address equivalence for the values of `hEvent` in each process.

4. To cleanup, first close the pool handle in the receiving process:

```
zeEventDestroy(hEvent);
zeEventPoolCloseIpcHandle(&hEventPool);
```

5. Finally, free the event pool handle in the sending process:

```
zeEventDestroy(hEvent);
zeEventPoolDestroy(hEventPool);
```

2.7.5 Peer-to-Peer Access and Queries

Devices may be linked together within a node by a scale-up fabric and depending on the configuration, the fabric can support atomics, compute kernel remote access, and data copies.

The following Peer-to-Peer functionalities are provided through the API:

- Check for existence of peer-to-peer fabric between two devices.
  - ::zeDeviceCanAccessPeer
- Query remote memory access and atomic capabilities for peer-to-peer
  - ::zeDeviceGetP2PProperties
- Copy data between devices over peer-to-peer fabric.
  - ::zeCommandListAppendMemoryCopy

CHAPTER

THREE

INITIALIZATION

The driver must be initialized by calling ::zetInit after calling ::zeInit and before calling any other experimental function. Simultaneous calls to ::zetInit are thread-safe.

CHAPTER

FOUR

API TRACING

4.1 Introduction

API tracing provides a way for tools to recieve notifications of API calls made by an applicaton. The callbacks provide direct access to the input and output parameters for viewing and modification. Tools may also use these notifications as triggers to block and inject new API calls into the command stream, such as metrics.

4.2 Registration

Tools may independently register for enter and exist callbacks for individual API calls, per driver instance.

- ::zetTracerSetPrologues is used to specify all the enter callbacks
- ::zetTracerSetEpilogues is used to specify all the exist callbacks
- If the value of a callback is nullptr, then it will be ignored.

The callbacks are defined as a collection of per-API function pointers, with the following parameters:

- params : a structure capturing pointers to the input and output parameters of the current instance
- result : the current value of the return value
- pTracerUserData : the user's pointer for the tracer's data
- ppTracerInstanceUserData : a per-tracer, per-instance storage location; typically used for passing data from the prologue to the epilogue

Note: since the creation of a tracer requires a device, on first glance it appears that ::zeInit, ::zeDriverGet and ::zeDeviceGet are not traceable. However, these APIs **are** traceable for all calls subsequent from the creation and enabling of the tracer itself.

4.3 Enabling/Disabling and Destruction

The tracer is created in a disabled state and must be explicitly enabled by calling ::zetTracerSetEnabled. The implementation guaranteed that prologues and epilogues will always be executed in pairs; i.e.

- if the prologue was called then the epilogue is guaranteed to be called, even if another thread disabled the tracer between execution
- if the prologue was not called then the epilogue is guaranteed not to be called, even if another thread enabled the tracer between execution

The tracer should be disabled by the application before the tracer is destoryed. If multiple threads are in-flight, then it is still possbile that callbacks will continue to execute even after the tracer is disabled; specifically due to the pairing rules above. Due to the complexity involved in ensuring no threads are still or will be executing a callback even after its been disabled, the implementation will stall and wait for any outstanding threads during ::zetTracerDestroy.

The following pseudo-code demonstrates a basic usage of API tracing:

```
typedef struct _my_tracer_data_t
{
    uint32_t instance;
} my_tracer_data_t;

typedef struct _my_instance_data_t
{
    clock_t start;
} my_instance_data_t;

void OnEnterCommandListAppendLaunchKernel(
    ze_command_list_append_launch_function_params_t* params,
    ze_result_t result,
    void* pTracerUserData,
    void** ppTracerInstanceUserData )
{
    my_instance_data_t* instance_data = malloc( sizeof(my_instance_data_t) );
    *ppTracerInstanceUserData = instance_data;

    instance_data->start = clock();
}

void OnExitCommandListAppendLaunchKernel(
    ze_command_list_append_launch_function_params_t* params,
    ze_result_t result,
    void* pTracerUserData,
    void** ppTracerInstanceUserData )
{
    clock_t end = clock();

    my_tracer_data_t* tracer_data = (my_tracer_data_t*)pTracerUserData;
    my_instance_data_t* instance_data = *(my_instance_data_
↪t**)ppTracerInstanceUserData;

    float time = 1000.f * ( end - instance_data->start ) / CLOCKS_PER_SEC;
    printf("zeCommandListAppendLaunchKernel #%d takes %.4f ms\n", tracer_data->
↪instance++, time);

    free(instance_data);
}

void TracingExample( ... )
{
    my_tracer_data_t tracer_data = {};
    zet_tracer_desc_t tracer_desc;
    tracer_desc.version = ZET_TRACER_DESC_VERSION_CURRENT;
    tracer_desc.pUserData = &tracer_data;
    zet_tracer_handle_t hTracer;
    zetTracerCreate(hDevice, &tracer_desc, &hTracer);

    // Set all callbacks
```

(continues on next page)

(continued from previous page)

```
    zet_core_callbacks_t prologCbs = {};
    zet_core_callbacks_t epilogCbs = {};
    prologCbs.CommandList.pfnAppendLaunchFunction =␣
↪OnEnterCommandListAppendLaunchKernel;
    epilogCbs.CommandList.pfnAppendLaunchFunction =␣
↪OnExitCommandListAppendLaunchKernel;

    zetTracerSetPrologues(hTracer, &prologCbs);
    zetTracerSetEpilogues(hTracer, &epilogCbs);

    zetTracerSetEnabled(hTracer, true);

    zeCommandListAppendLaunchKernel(hCommandList, hFunction, &launchArgs, nullptr, 0,␣
↪nullptr);
    zeCommandListAppendLaunchKernel(hCommandList, hFunction, &launchArgs, nullptr, 0,␣
↪nullptr);
    zeCommandListAppendLaunchKernel(hCommandList, hFunction, &launchArgs, nullptr, 0,␣
↪nullptr);

    zetTracerSetEnabled(hTracer, false);
    zetTracerDestroy(hTracer);
}
```

CHAPTER

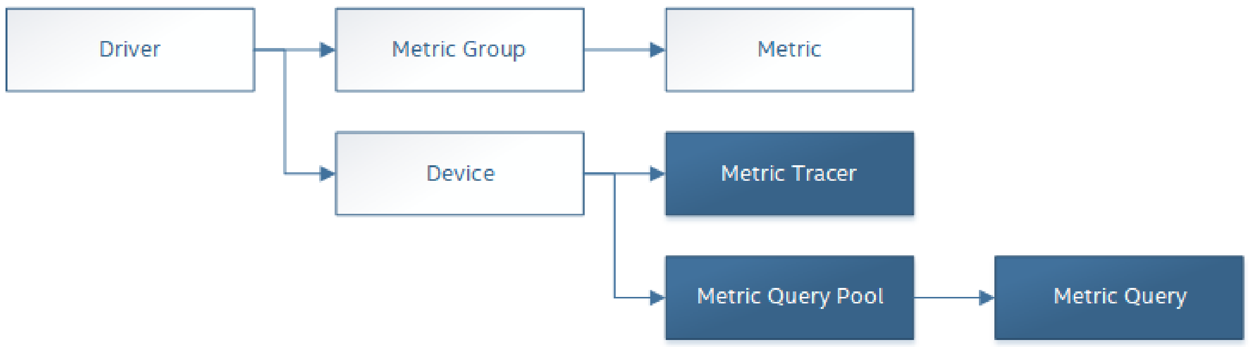
FIVE

METRICS

5.1 Introduction

Devices provide programmable infrastructure designed to support performance debugging. The API described in this document provides access to these device metrics.

The following diagram illustrates the relationship between the metric objects described in this document.



Most of the detailed metrics require the device to be properly programmed before use. It is important to understand that the device programming is in most cases global. This generally means that if a software tool or an application is using the metrics then no other application can reliably use the same device resources.

The use of some metrics may negatively impact the performance of the device. The intention of this API is to support performance debug and it is not advised to use it in regular execution.

5.2 Metric Groups

The device infrastucture consists of non-programmable, pre-defined set of counters, and a programmable network of connections that work with a separate set of counters as well as other types of counters. For sake of simplicity, the smallest unit of configuration is a Metric Group. Metric Groups are sets of metrics that provide certain perspective on workload's performance. The groups aggregate metrics, define device programming and available collection methods. An application may choose to collect data from a number of Metric Groups provided that they all belong to different domains. *Domains* are used as a software representation of independent device resources that can safely be used concurrently.

5.2.1 Sampling Types

Sampling types are a software representation of device capabilities in terms of reading metric values. Each Metric Group provides information which sampling types it supports. There are separate sets of APIs supporting each of the sampling types *Time-based* and *Event-based*.

All available sampling types are defined in ::zet_metric_group_sampling_type_t.

- Information about supported sampling types for a given Metric Group is provided in ::zet_metric_group_properties_t.samplingType.
- It's possible that a device provides multiple Metric Groups with the same names but different sampling types.
- When enumerating, it's important to choose a Metric Group which supports the desired sampling type.

5.2.2 Domains

Every Metric Group belongs to a given domain (::zet_metric_group_properties_t.domain).

- The Metric Group typically define a uniform device counter configuration used for measurements.
- Each domain represents an exclusive resource used by the Metric Group.
- It's possible to simultaneously gather data for two different Metric Groups, only if they belong to a different domain i.e. Metric Groups that can be collected concurrently will have different domain values.

5.3 Enumeration

All available metrics are organized into Metric Groups.

- During data collection, data for the whole Metric Group is gathered.
- The list of available Metric Groups and their Metrics is device-specific.

The following APIs provide all the information needed for identification and usage.

- Metric Group properties are accessed through function ::zetMetricGroupGetProperties, returning ::zet_metric_group_properties_t.
- Metric properties are accessed through function ::zetMetricGetProperties, returning ::zet_metric_properties_t.

A common tool flow is to enumerate metrics looking for a specific Metric Group. Depending on the metrics required for a specific scenario a tool may choose to run the workload multiple times, recording different set of Metric Groups each time. Usually care must be taken to ensure run-to-run stability and result repeatability if metrics from different runs are meant to be used together. When enumerating Metric tree to find a desired Metric Group, it's important to know in advance which sampling type it will be used.

To enumerate through the Metric tree:

1. Call ::zetMetricGroupGet to obtain Metric Group count.
2. Call ::zetMetricGroupGet to obtain all Metric Groups.
3. Iterate over all available Metric Groups.
    - At this point it's possible to check e.g. Metric Group name, domain or sampling type.
    - Metric Group names may not be unique.
4. For each Metric Group obtain their Metric count calling ::zetMetricGroupGetProperties with Metric Group handle (::zet_metric_group_handle_t) and checking zet_metric_group_properties_t.metricCount.

5. Iterate over available Metrics using ::zetMetricGet with parent Metric Group (::zet_metric_group_handle_t).
6. Check Metric properties (e.g. name, description) calling ::zetMetricGetProperties with parent Metric (::zet_metric_handle_t).

The following pseudo-code demonstrates a basic enumeration over all available metric groups and their metrics. Additionally, it returns a metric group with a chosen name and sampling type. Similar code could be used for selecting a preferred metric group for a specific type of measurements.

```
ze_result_t FindMetricGroup( ze_device_handle_t hDevice,
                             char* pMetricGroupName,
                             uint32_t desiredSamplingType,
                             zet_metric_group_handle_t* phMetricGroup )
{
    // Obtain available metric groups for the specific device
    uint32_t metricGroupCount = 0;
    zetMetricGroupGet( hDevice, &metricGroupCount, nullptr );

    zet_metric_group_handle_t* phMetricGroups = malloc(metricGroupCount * sizeof(zet_
↪metric_group_handle_t));
    zetMetricGroupGet( hDevice, &metricGroupCount, phMetricGroups );

    // Iterate over all metric groups available
    for( i = 0; i < metricGroupCount; i++ )
    {
        // Get metric group under index 'i' and its properties
        zet_metric_group_properties_t metricGroupProperties;
        zetMetricGroupGetProperties( phMetricGroups[i], &metricGroupProperties );

        printf("Metric Group: %s\n", metricGroupProperties.name);

        // Check whether the obtained metric group supports the desired sampling type
        if((metricGroupProperties.samplingType & desiredSamplingType) ==␣
↪desiredSamplingType)
        {
            // Check whether the obtained metric group has the desired name
            if( strcmp( pMetricGroupName, metricGroupProperties.name ) == 0 )
            {
                *phMetricGroup = phMetricGroups[i];
                break;
            }
        }
    }

    free(phMetricGroups);
}
```

5.4 Configuration

Use the ::zetDeviceActivateMetricGroups API call to configure the device for data collection.

- Subsequent calls to the function will disable device programming for the metric groups not selected for activation.
- To avoid undefined results only call the ::zetDeviceActivateMetricGroups between experiments i.e. while not collecting data.

Programming restrictions:

- Any combination of metric groups can be configured simultaneously provided that all of them have different ::zet_metric_group_properties_t.domain.
- MetricGroup must be active until ::zetMetricQueryGetData and ::zetMetricTracerClose.
- Conflicting Groups cannot be activated, in such case the call to ::zetDeviceActivateMetricGroups would fail.

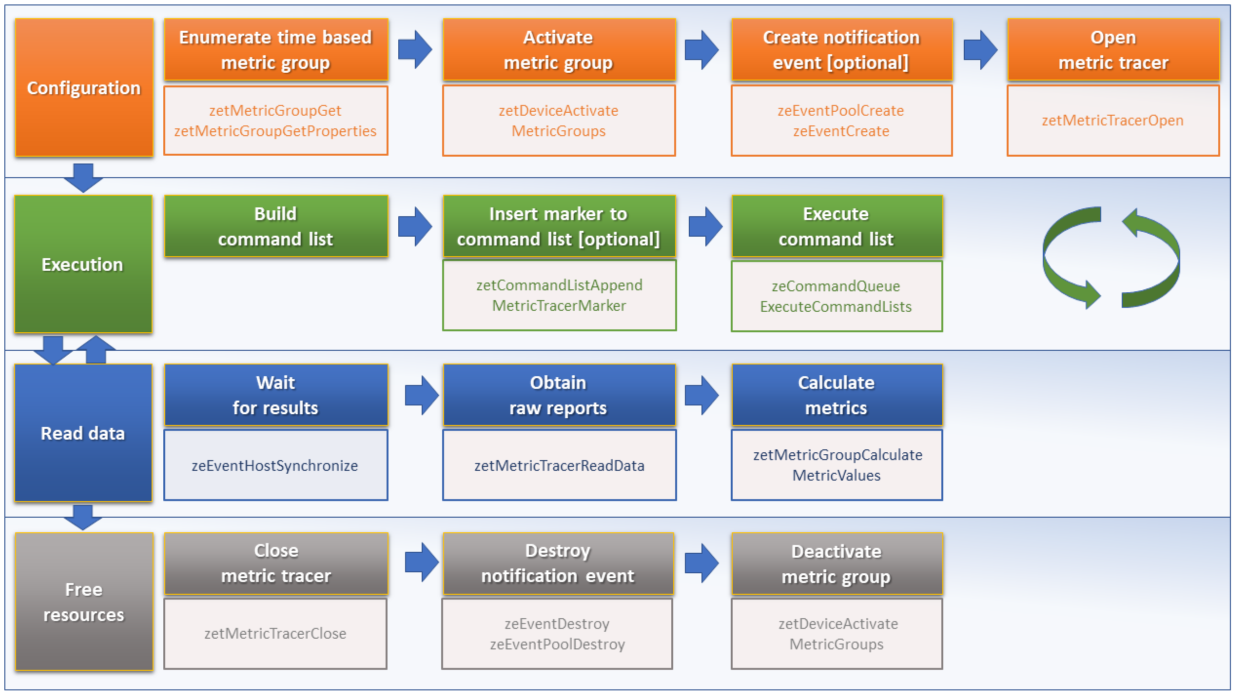
5.5 Collection

There are two modes of metrics collection supported: time-based and event-based.

- Time-based collection is using a timer as well as other events to store data samples. A metric tracer interface is the software interface for configuration and collection.
- Event-based metrics collection is based on a pair of Begin/End events appended to command lists. A metric query interface is the software interface for configuration and collection.

5.5.1 Metric Tracer

Time-based collection uses a simple Open, Wait, Read, Close scheme: - ::zetMetricTracerOpen opens the tracer. - ::zeEventHostSynchronize and ::zeEventQueryStatus can be used to wait for data. - ::zetMetricTracerReadData reads the data to be later processed by ::zetMetricGroupCalculateMetricValues. - ::zetMetricTracerClose closes the tracer.


Configuration
Enumerate time based metric group
zetMetricGroupGet zetMetricGroupGetProperties
Activate metric group
zetDeviceActivate MetricGroups
Create notification event [optional]
zeEventPoolCreate zeEventCreate
Open metric tracer
zetMetricTracerOpen
Execution
Build command list
Insert marker to command list [optional]
zetCommandListAppend MetricTracerMarker
Execute command list
zeCommandQueue ExecuteCommandLists
Read data
Wait for results
zeEventHostSynchronize
Obtain raw reports
zetMetricTracerReadData
Calculate metrics
zetMetricGroupCalculate MetricValues
Free resources
Close metric tracer
zetMetricTracerClose
Destroy notification event
zeEventDestroy zeEventPoolDestroy
Deactivate metric group
zetDeviceActivate MetricGroups

The following pseudo-code demonstrates a basic sequence for tracer-based collection:

```
ze_result_t TimeBasedUsageExample( ze_driver_handle_t hDriver,
                                    ze_device_handle_t hDevice )
{
    zet_metric_group_handle_t     hMetricGroup           = nullptr;
    ze_event_handle_t            hNotificationEvent     = nullptr;
    ze_event_pool_handle_t       hEventPool             = nullptr;
```

(continues on next page)

(continued from previous page)

```
    ze_event_pool_desc_t          eventPoolDesc           = {ZE_EVENT_POOL_DESC_VERSION_
↪CURRENT, ZE_EVENT_POOL_FLAG_DEFAULT , 1};
    ze_event_desc_t               eventDesc               = {ZE_EVENT_DESC_VERSION_
↪CURRENT};
    zet_metric_tracer_handle_t     hMetricTracer           = nullptr;
    zet_metric_tracer_desc_t       metricTracerDescriptor = {ZET_METRIC_TRACER_DESC_
↪VERSION_CURRENT};

    // Find a "ComputeBasic" metric group suitable for Time Based collection
    FindMetricGroup( hDevice, "ComputeBasic", ZET_METRIC_GROUP_SAMPLING_TYPE_TIME_
↪BASED, &hMetricGroup );

    // Configure the HW
    zetDeviceActivateMetricGroups( hDevice, 1 /* count */, &hMetricGroup );

    // Create notification event
    zeEventPoolCreate( hDriver, &eventPoolDesc, 1, &hDevice, &hEventPool );
    eventDesc.index  = 0;
    eventDesc.signal = XE_EVENT_SCOPE_FLAG_HOST;
    eventDesc.wait   = XE_EVENT_SCOPE_FLAG_HOST;
    zeEventCreate( hEventPool, &eventDesc, &hNotificationEvent );

    // Open metric tracer
    metricTracerDescriptor.samplingPeriod       = 1000;
    metricTracerDescriptor.notifyEveryNReports  = 32768;
    zetMetricTracerOpen( hDevice, hMetricGroup, &metricTracerDescriptor,␣
↪hNotificationEvent, &hMetricTracer );

    // Run your workload, in this example we assume the data for the whole experiment␣
↪fits in the device buffer
    Workload(hDevice);
    // Optionally insert markers during workload execution
    //zetCommandListAppendMetricTracerMarker( hCommandList, hMetricTracer, tool_
↪marker_value );

    // Wait for data, optional in this example since the whole workload has already␣
↪been executed by now
    //zeEventHostSynchronize( hNotificationEvent, 1000 /*timeout*/ );
    // reset the event if it fired

    // Read raw data
    size_t rawSize = 0;
    zetMetricTracerReadData( hMetricTracer, UINT32_MAX, &rawSize, nullptr );
    uint8_t* rawData = malloc(rawSize);
    zetMetricTracerReadData( hMetricTracer, UINT32_MAX, &rawSize, rawData );

    // Close metric tracer
    zetMetricTracerClose( hMetricTracer );
    zeEventDestroy( hNotificationEvent );
    zeEventPoolDestroy( hEventPool );

    // Deconfigure the device
    zetDeviceActivateMetricGroups( hDevice, 0, nullptr );

    // Calculate metric data
    CalculateMetricsExample( hMetricGroup, rawSize, rawData );
    free(rawData);
```

(continues on next page)

(continued from previous page)

```
}
```

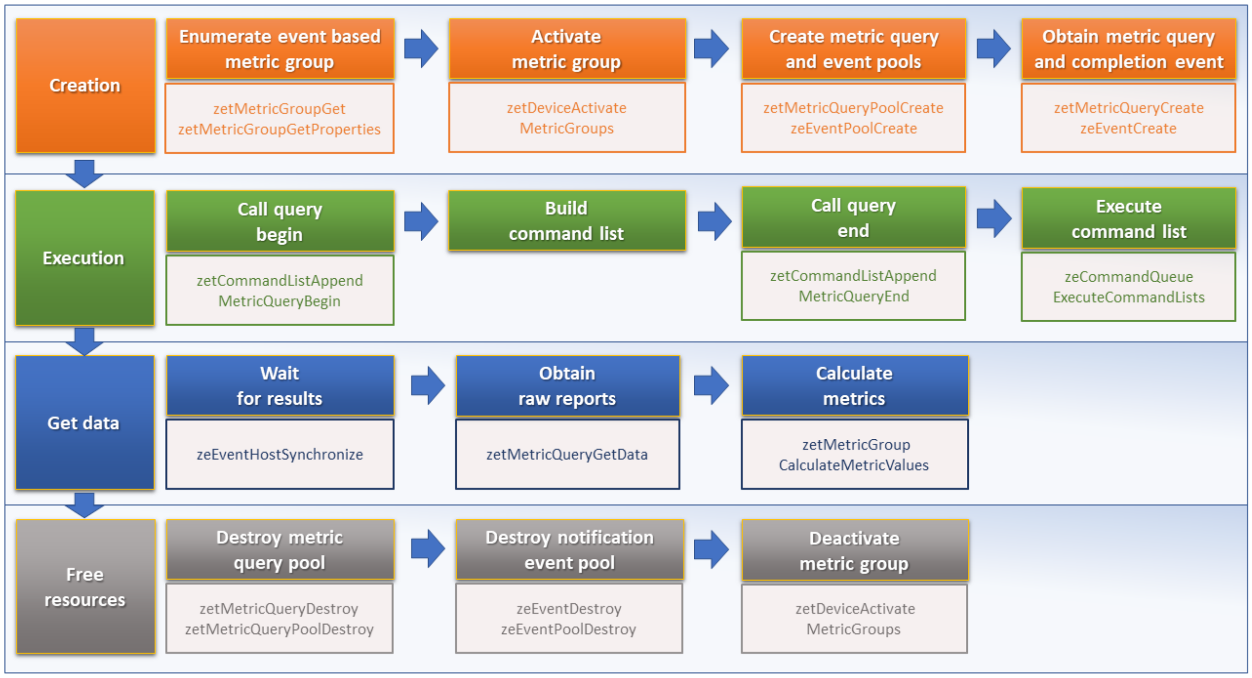
5.5.2 Metric Query

Event-based collection uses a simple Begin, End, GetData scheme:

- ::zetCommandListAppendMetricQueryBegin defines the start counting event
- ::zetCommandListAppendMetricQueryEnd defines the finish counting event
- ::zetMetricQueryGetData reads the raw data to be later processed by ::zetMetricGroupCalculateMetricValues.

Typically, multiple queries are used and recycled to characterize a workload. A Query Pool is used to efficiently use and reuse device memory for multiple queries.

- ::zetMetricQueryPoolCreate creates a pool of homogeneous queries.
- ::zetMetricQueryPoolDestroy frees the pool. The application must ensure no queries within the pool are in-use before freeing the pool.
- ::zetMetricQueryCreate obtains a handle to a unique location in the pool.
- ::zetMetricQueryReset allows for low-cost recycling of a location in the pool.


Creation
Enumerate event based metric group
zetMetricGroupGet
zetMetricGroupGetProperties
Activate metric group
zetDeviceActivate MetricGroups
Create metric query and event pools
zetMetricQueryPoolCreate
zeEventPoolCreate
Obtain metric query and completion event
zetMetricQueryCreate
zeEventCreate
Execution
Call query begin
zetCommandListAppend MetricQueryBegin
Build command list
Call query end
zetCommandListAppend MetricQueryEnd
Execute command list
zeCommandQueue ExecuteCommandLists
Get data
Wait for results
zeEventHostSynchronize
Obtain raw reports
zetMetricQueryGetData
Calculate metrics
zetMetricGroup CalculateMetricValues
Free resources
Destroy metric query pool
zetMetricQueryDestroy
zetMetricQueryPoolDestroy
Destroy notification event pool
zeEventDestroy
zeEventPoolDestroy
Deactivate metric group
zetDeviceActivate MetricGroups

The following pseudo-code demonstrates a basic sequence for query-based collection:

```
ze_result_t MetricQueryUsageExample( ze_driver_handle_t hDriver,
                                     ze_device_handle_t hDevice )
{
    zet_metric_group_handle_t      hMetricGroup          = nullptr;
    ze_event_handle_t             hCompletionEvent      = nullptr;
    ze_event_pool_desc_t          eventPoolDesc         = {ZE_EVENT_POOL_DESC_VERSION_
↪CURRENT};
```

(continues on next page)

(continued from previous page)

```
    ze_event_desc_t                eventDesc              = {ZE_EVENT_DESC_VERSION_
↪CURRENT};
    ze_event_pool_handle_t         hEventPool             = nullptr;
    zet_metric_query_pool_handle_t hMetricQueryPool        = nullptr;
    zet_metric_query_handle_t      hMetricQuery            = nullptr;
    zet_metric_query_pool_desc_t   queryPoolDesc           = {ZET_METRIC_QUERY_POOL_
↪DESC_VERSION_CURRENT};

    // Find a "ComputeBasic" metric group suitable for Event Based collection
    FindMetricGroup( hDevice, "ComputeBasic", ZET_METRIC_GROUP_SAMPLING_TYPE_EVENT_
↪BASED, &hMetricGroup );

    // Configure HW
    zetDeviceActivateMetricGroups( hDevice, 1 /* count */, &hMetricGroup );

    // Create metric query pool & completion event
    queryPoolDesc.flags         = ZET_METRIC_QUERY_POOL_FLAG_PERFORMANCE;
    queryPoolDesc.count         = 1000;
    zetMetricQueryPoolCreate( hDevice, hMetricGroup, &queryPoolDesc, &
↪hMetricQueryPool );
    eventPoolDesc.flags = ZE_EVENT_POOL_FLAG_DEFAULT;
    eventPoolDesc.count = 1000;
    zeEventPoolCreate( hDriver, &eventPoolDesc, 1, &hDevice, &hEventPool );

    // Write BEGIN metric query to command list
    zetMetricQueryCreate( hMetricQueryPool, 0 /*slot*/, &hMetricQuery );
    zetCommandListAppendMetricQueryBegin( hCommandList, hMetricQuery );

    // build your command list

    // Write END metric query to command list, use an event to determine if the data␣
↪is available
    eventDesc.index  = 0;
    eventDesc.signal = XE_EVENT_SCOPE_FLAG_HOST;
    eventDesc.wait   = XE_EVENT_SCOPE_FLAG_HOST;
    zeEventCreate( hEventPool, &eventDesc, &hCompletionEvent);
    zetCommandListAppendMetricQueryEnd( hCommandList, hMetricQuery, hCompletionEvent␣
↪);

    // use zeCommandQueueExecuteCommandLists( , , , ) to submit your workload to the␣
↪device

    // Wait for data
    zeEventHostSynchronize( hCompletionEvent, 1000 /*timeout*/ );

    // Read raw data
    size_t rawSize = 0;
    zetMetricQueryGetData( hMetricQuery, &rawSize, nullptr );
    uint8_t* rawData = malloc(rawSize);
    zetMetricQueryGetData( hMetricQuery, &rawSize, rawData );

    // Free the resources
    zeEventDestroy( hCompletionEvent );
    zeEventPoolDestroy( hEventPool );
    zetMetricQueryPoolDestroy( hMetricQueryPool );

    // Deconfigure HW
```

(continues on next page)

(continued from previous page)

```
    zetDeviceActivateMetricGroups( hDevice, 0, nullptr );

    // Calculate metric data
    CalculateMetricsExample( hMetricGroup, rawSize, rawData );
    free(rawData);
}
```

5.6 Calculation

Both MetricTracer and MetricQueryPool collect the data in device specific, raw form that is not suitable for application processing. To calculate metric values use ::zetMetricGroupCalculateMetricValues.

The following pseudo-code demonstrates a basic sequence for metric calculation and interpretation:

```
ze_result_t CalculateMetricsExample( zet_metric_group_handle_t hMetricGroup,
                                       size_t rawSize, uint8_t* rawData )
{
    // Calculate metric data
    uint32_t numMetricValues = 0;
    zetMetricGroupCalculateMetricValues( hMetricGroup, rawSize, rawData, &
↪numMetricValues, nullptr );
    zet_typed_value_t* metricValues = malloc( numMetricValues * sizeof(zet_typed_
↪value_t) );
    zetMetricGroupCalculateMetricValues( hMetricGroup, rawSize, rawData, &
↪numMetricValues, metricValues );

    // Obtain available metrics for the specific metric group
    uint32_t metricCount = 0;
    zetMetricGet( hMetricGroup, &metricCount, nullptr );

    zet_metric_handle_t* phMetrics = malloc(metricCount * sizeof(zet_metric_handle_
↪t));
    zetMetricGet( hMetricGroup, &metricCount, phMetrics );

    // Print metric results
    uint32_t numReports = numMetricValues / metricCount;
    for( uint32_t report = 0; report < numReports; ++report )
    {
        printf("Report: %d\n", report);

        for( uint32_t metric = 0; metric < metricCount; ++metric )
        {
            zet_typed_value_t data = metricValues[report * metricCount + metric];

            zet_metric_properties_t metricProperties;
            zetMetricGetProperties( phMetrics[ metric ], &metricProperties );

            printf("Metric: %s\n", metricProperties.name );

            switch( data.type )
            {
            case ZET_VALUE_TYPE_UINT32:
                printf(" Value: %lu\n", data.value.ui32 );
                break;
```

(continues on next page)

(continued from previous page)

```
            case ZET_VALUE_TYPE_UINT64:
                printf(" Value: %llu\n", data.value.ui64 );
                break;
            case ZET_VALUE_TYPE_FLOAT32:
                printf(" Value: %f\n", data.value.fp32 );
                break;
            case ZET_VALUE_TYPE_FLOAT64:
                printf(" Value: %f\n", data.value.fp64 );
                break;
            case ZET_VALUE_TYPE_BOOL8:
                if( data.value.ui32 )
                    printf(" Value: true\n" );
                else
                    printf(" Value: false\n" );
                break;
            default:
                break;
            };
        }
    }

    free(metricValues);
    free(phMetrics);
}
```

CHAPTER

SIX

PROGRAM INSTRUMENTATION

6.1 Introduction

The program instrumentation APIs provide tools a basic framework for low-level profiling of device programs, by allowing direct instrumentation of those programs. These capabilities, in combination with those already provided, in combination with API tracing, are sufficient for more advanced frameworks to be developed independently.

There are two type of instrumentation available:

1. Inter-Function Instrumentation - intercepting and redirecting function calls
2. Intra-Function Instrumentation - injecting new instructions within a function

6.2 Inter-Function Instrumentation

The following capabilities allow for a tool to intercept and redirect function calls:

- Inter-module function calls - the ability to call functions between different modules; e.g., the application's module and a tool's module
- *API-Tracing*

For example, a tool may use API Tracing in any of the following ways:

- ::zeModuleCreate - replace a module handle with instrumented module handle for all functions
- ::zeKernelCreate - replace a kernel handle with instrumented kernel handle for all call sites
- ::zeModuleGetFunctionPointer - replace a function pointer with instrumented function pointer for all call sites
- ::zeCommandListAppendLaunchKernel - replace a kernel handle with instrumented kernel handle at call site

6.3 Intra-Function Instrumentation

The following capabilities allow for a tool to inject instructions within a kernel:

- ::zetModuleGetDebugInfo - allows a tool to query standard debug info for an application's module
- ::zetKernelGetProfileInfo - allows a tool query detailed information on aspects of a kernel
- ::zeModuleGetNativeBinary - allows for a tool to retrieve the native binary of the application's module, instrument it, then create a new module using the intrumented version
- *API-Tracing* - same usage as Inter-Function Instrumentation above

6.3.1 Compilation

A module must be compiled with foreknowledge that instrumentation will be performed in order for the compiler to generate the proper profiling meta-data. Therefore, when the instrumentation layer is enabled, a new build flag is supported: "-zet-profile-flags", where "" must be a combination of ::zet_profile_flag_t, in hexidecimal.

As an example, a tool could use API Tracing to inject this build flag on each ::zeModuleCreate call that the tool wishes to instrument. In another example, a tool could recompile a Module using the build flag and use API Tracing to replace the application's Module handle with it's own.

6.3.2 Instrumentation

Once the module has been compiled with instrumentation enabled, a tool may use ::zetModuleGetDebugInfo and ::zetKernelGetProfileInfo in order to decode the application's instructions and register usage for each function in the module.

If a tool requires additional functions to be used, it may create other module(s) and use ::zeModuleGetFunctionPointer to call functions between the application and tool modules. A tool may use ::zeModuleGetFunctionPointer to retrieve the Host and device address of each function in the module.

There are no APIs provided for the actual instrumentation. Instead this is left up to the tool itself to decode the application module's native binary and inject native instructions. This model prevents the instrumentation from being manipulated by the compiler.

6.3.3 Execution

If a tool requires changing the address of an application's function, then it should use API Tracing; for example, ::zeModuleGetFunctionPointer and all flavors of ::zeCommandListAppendLaunchKernel.

CHAPTER

SEVEN

PROGRAM DEBUG

7.1 Introduction

The program debug APIs provide tools a basic framework for inserting breakpoints and accessing register values of device programs, as they are executing on the device.

(more details coming soon…)

CHAPTER EIGHT

INTRODUCTION

Sysman is the System Resource Management library used to monitor and control the power and performance of accelerator devices.

CHAPTER

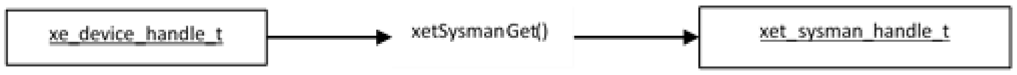
NINE

HIGH-LEVEL OVERVIEW

9.1 Initialization

An application wishing to manage power and performance for devices first needs to use the Level0 Core API to enumerate through available accelerator devices in the system and select those of interest.

For each selected device handle, applications use the function ::zetSysmanGet() to get an **Sysman handle** to manage system resources of the device.



There is a unique handle for each device. Multiple threads can use the handle. If concurrent accesses are made to the same device property through the handle, the last request wins.

The pseudo code below shows how to enumerate the GPU devices in the system and create Sysman handles for them:

```
function main( ... )
    if ( (zeInit(ZE_INIT_FLAG_NONE) != ZE_RESULT_SUCCESS) or
         (zetInit(ZE_INIT_FLAG_NONE) != ZE_RESULT_SUCCESS) )
        output("Can't initialize the API")
    else
        # Discover all the drivers
        uint32_t driversCount = 0
        zeDriverGet(&driversCount, nullptr)
        ze_driver_handle_t* allDrivers = allocate(driversCount * sizeof(ze_driver_
↪handle_t))
        zeDriverGet(&driversCount, allDrivers)

        ze_driver_handle_t hDriver = nullptr
        for(i = 0 .. driversCount-1)
            # Discover devices in a driver
            uint32_t deviceCount = 0
            zeDeviceGet(allDrivers[i], &deviceCount, nullptr)

            ze_device_handle_t* allDevices =
                allocate_memory(deviceCount * sizeof(ze_device_handle_t))
            zeDeviceGet(allDrivers[i], &deviceCount, allDevices)

            for(devIndex = 0 .. deviceCount-1)
                ze_device_properties_t device_properties
```

(continues on next page)

(continued from previous page)

```
                zeDeviceGetProperties(allDevices[devIndex], &device_properties)
                if(ZE_DEVICE_TYPE_GPU != device_properties.type)
                    next
                # Create Sysman handle
                zet_sysman_handle_t hSysmanDevice
                ze_result_t res = zetSysmanGet(hDevice, ZET_SYSMAN_VERSION_CURRENT, &
↪hSysmanDevice)
                if (res == ZE_RESULT_SUCCESS)
                    # Start using hSysmanDevice to manage the device
                else
                    output("ERROR: Can't initialize system resource management for␣
↪this device")

    free_memory(...)
```

9.2 Global device management

The following operations are provided to access overall device information and control aspects of the entire device:

- Get device UUID, deviceID, number of sub-devices
- Get Brand/model/vendor name
- Query the information about processes using this device
- Get/set scheduler mode and properties
- Reset device
- Query if the device has been repaired
- PCI information:
  - Get configured bars
  - Get maximum supported bandwidth
  - Query current speed (GEN/no. lanes)
  - Query current throughput
  - Query packet retry counters

The full list of available functions is described *below*.

9.3 Device component management

Aside from management of the global properties of a device, there are many device components that can be managed to change the performance and/or power configuration of the device. Similar components are broken into **classes** and each class has a set of operations that can be performed on them.

For example, devices typically have one or more frequency domains. The Sysman API exposes a class for frequency and an enumeration of all frequency domains that can be managed.

The table below summarizes the classes that provide device queries and an example list of components that would be enumerated for a device with two sub-devices. The table shows the operations (queries) that will be provided for all components in each class.

Class	Components	Operations
Power	Package: powerSub-device 0: Total powerSub-device 1: Total power	Get energy consumption
Frequency	Sub-device 0: GPU frequencySub-device 0: Memory frequencySub-device 1: GPU frequencySub-device 1: Memory frequency	List available frequenciesSet frequency rangeGet frequenciesGet throttle reasonsGet throttle time
Engines	Sub-device 0: All enginesSub-device 0: Compute enginesSub-device 0: Media enginesSub-device 1: All enginesSub-device 1: Compute enginesSub-device 1: Media engines	Get busy time
Firmware	Sub-device 0: Enumerates each firmwareSub-device 1: Enumerates each firmware	Get firmware name and versionVerify firmware checksum
Memory	Sub-device 0: Memory module Sub-device 1: Memory module	Get maximum supported bandwidthGet current allocation sizeGet current bandwidth
Fabric-Port	Enumerates each portSub-device 1: Enumerates each port	configuration (UP/DOWN)Get physical link detailsGet port health (green/yellow/red/bla ck)Get remote port UUIDGet port max rx/tx speedGet port current rx/tx bandwidth
Temperature	Package: temperatureSub-device 0: GPU temperatureSub-device 0: Memory temperatureSub-device 1: GPU temperatureSub-device 1: Memory temperature	Get current temperature sensor reading
PSU	Package: Power supplies	Get details about the power supplyQuery current state (temperature,current, fan)
Fan	Package: Fans	Get details (max fan speed)Get config (fixed fan speed, temperature-speed table)Query current fan speed
LED	Package: LEDs	Get details (supports RGB configuration)Query current state (on,color)
RAS	Sub-device 0: One set of RAS error countersSub-device 1: One set of RAS error counters	Read RAS total correctable and uncorrectable error counter.Read breakdown of errors by category:- no. resets- no. programming errors- no. driver errors- no. compute errors- no. cache errors- no. memory errors- no. PCI errors- no. fabric port errors- no. display errors- no. non-compute errors
Diagnostics	Package: SCAN test suitePackage: ARRAY test suite	Get list of all diagnostics tests in the test suite

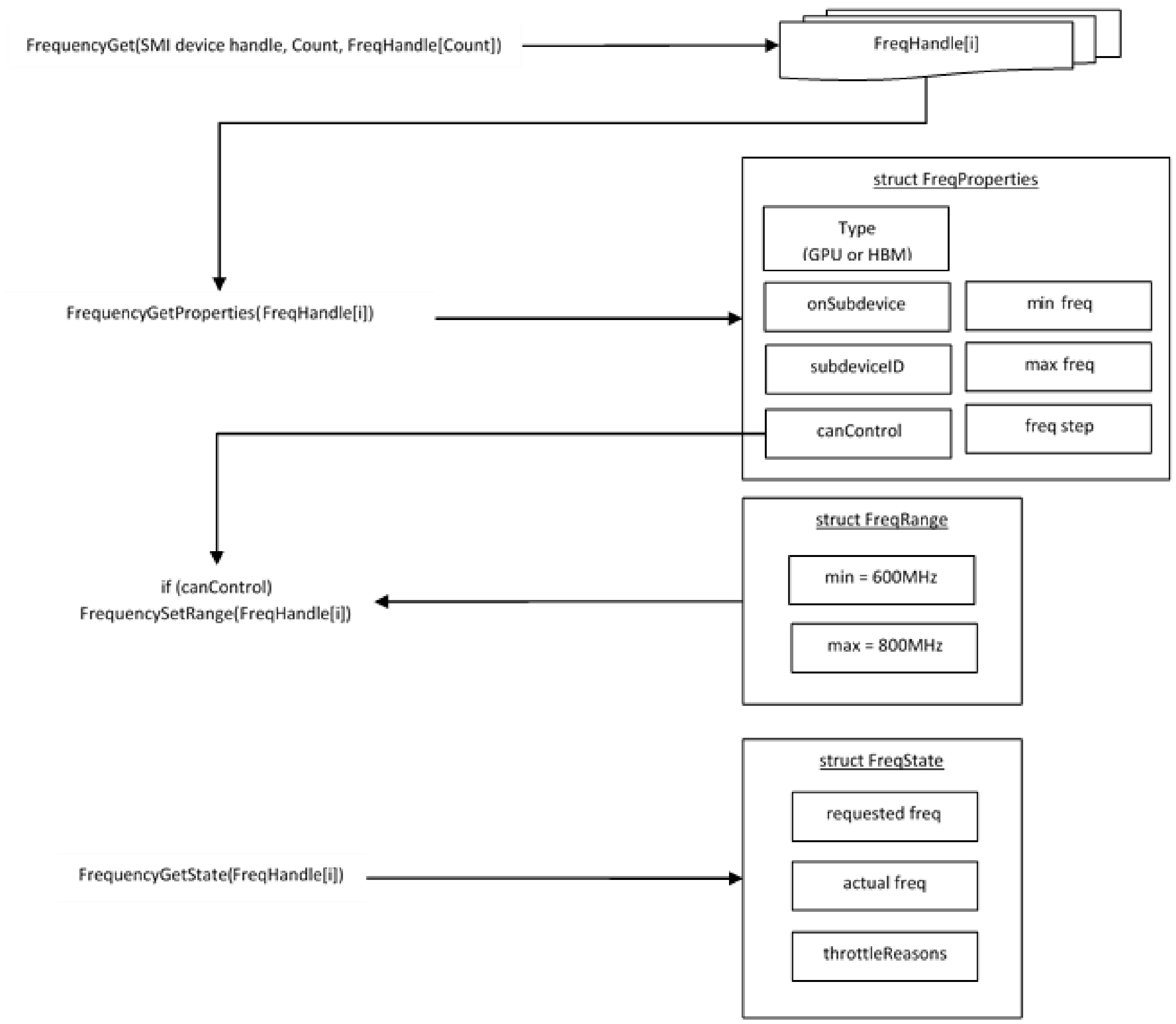
The table below summarizes the classes that provide device controls and an example list of components that would be enumerated for a device with two sub-devices. The table shows the operations (controls) that will be provided for all components in each class.

Class	Components	Operations
Power	Package: power	Set sustained power limitSet burst power limitSet peak power limit
Frequency	Sub-device 0: GPU frequencySub-device 0: Memory frequencySub-device 1: GPU frequencySub-device 1: Memory frequency	Set frequency range
Standby	Sub-device 0: Control entire sub-deviceSub-device 1: Control entire sub-device	Disable opportunistic standby
Firmware	Sub-device 0: Enumerates each firmwareSub-device 1: Enumerates each firmware	Flash new firmware
Fabric port	Sub-device 0: Control each portSub-device 1: Control each port	Configure port UP/DOWNTurn beaconing ON/OFF
Fan	Package: Fans	Set config (fixed speed, temperature-speed table)
LED	Package: LEDs	Turn LED on/off and set color where applicable
Diagnostics	SCAN test suiteARRAY test suite	Run all or a subset of diagnostic tests in the test suite

9.4 Device component enumeration

The Sysman API provides functions to enumerate all components in a class that can be managed.

For example, there is a frequency class which is used to control the frequency of different parts of the device. On most devices, the enumerator will provide two handles, one to control the GPU frequency and one to enumerate the device memory frequency. This is illustrated in the figure below:



In the C API, each class is associated with a unique handle type (e.g. ::zet_sysman_freq_handle_t refers to a frequency component). In the C++ API, each class is a C++ class (e.g. An instance of the class ::zet::SysmanFrequency refers to a frequency component).

The pseudo code below shows how to use the Sysman API to enumerate all GPU frequency components and fix each to a specific frequency if this is supported:

```
function FixGpuFrequency(zet_sysman_handle_t hSysmanDevice, double FreqMHz)
    uint32_t numFreqDomains
    if ((zetSysmanFrequencyGet(hSysmanDevice, &numFreqDomains, NULL) == ZE_RESULT_
↪SUCCESS))
        zet_sysman_freq_handle_t* pFreqHandles =
            allocate_memory(numFreqDomains * sizeof(zet_sysman_freq_handle_t))
        if (zetSysmanFrequencyGet(hSysmanDevice, &numFreqDomains, pFreqHandles) == ZE_
↪RESULT_SUCCESS)
            for (index = 0 .. numFreqDomains-1)
                zet_freq_properties_t props
                if (zetSysmanFrequencyGetProperties(pFreqHandles[index], &props) ==␣
↪ZE_RESULT_SUCCESS)
                    # Only change the frequency of the domain if:
```

(continues on next page)

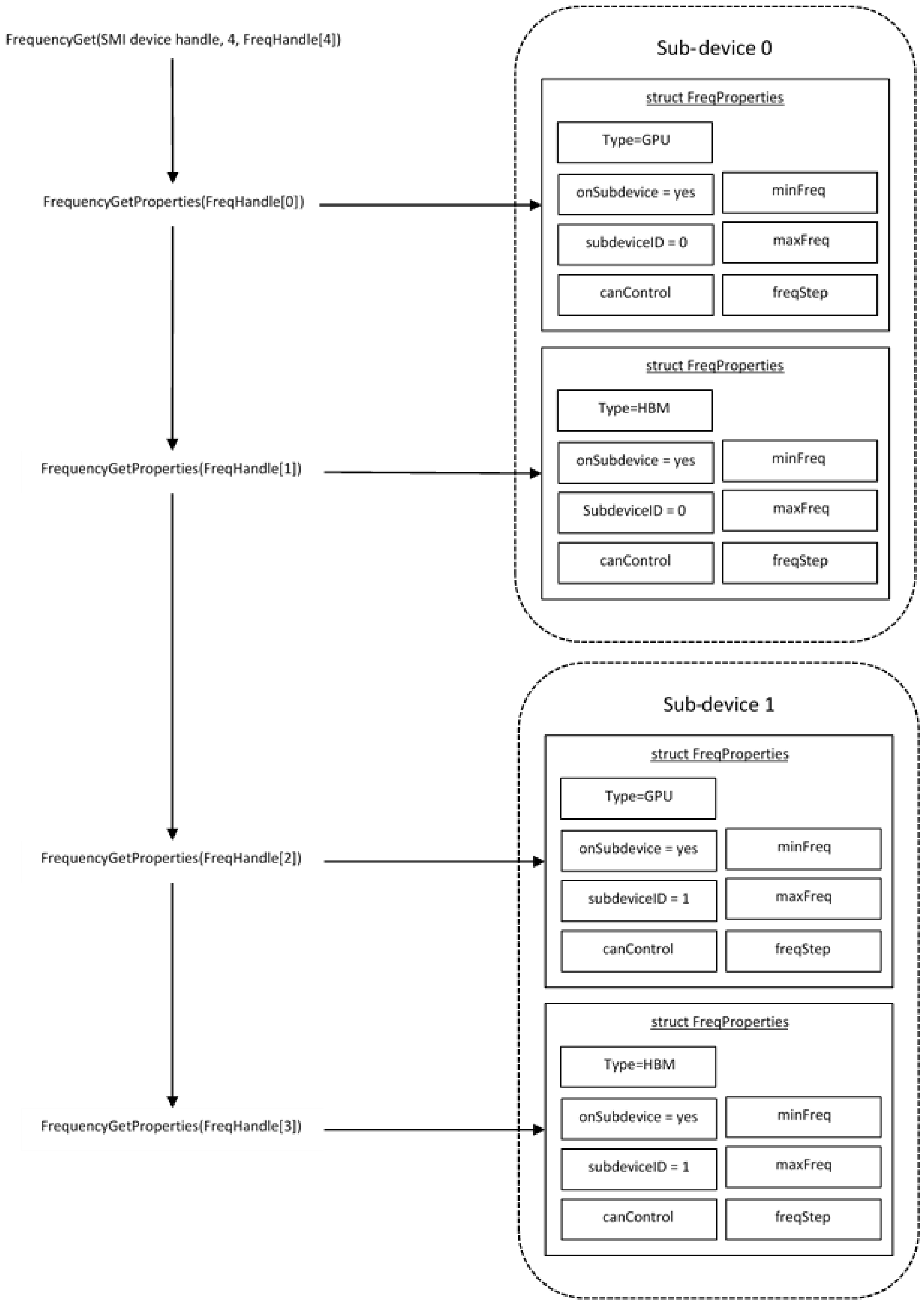
(continued from previous page)

```
                    # 1. The domain controls a GPU accelerator
                    # 2. The domain frequency can be changed
                    if (props.type == ZET_FREQ_DOMAIN_GPU
                        and props.canControl)
                            # Fix the frequency
                            zet_freq_range_t range
                            range.min = FreqMHz
                            range.max = FreqMHz
                            zetSysmanFrequencySetRange(pFreqHandles[index], &range)
    free_memory(...)
```

9.5 Sub-device management

A Sysman handle cannot be created for a sub-device - ::zetSysmanGet() will return error ::ZE_RESULT_ERROR_INVALID_ARGUMENT if a device handle for a sub-device is passed to this function. Instead, the enumerator for device components will return a list of components that are located in each sub-device. Properties for each component will indicate in which sub-device it is located. If software wishing to manage components in only one sub-device should filter the enumerated components using the sub-device ID (see ::ze_device_properties_t.subdeviceId).

The figure below shows the frequency components that will be enumerated on a device with two sub-devices where each sub-device has a GPU and device memory frequency control:



The pseudo code below shows how to fix the GPU frequency on a specific sub-device (notice the additional sub-device check):

```
function FixSubdeviceGpuFrequency(zet_sysman_handle_t hSysmanDevice, uint32_t␣
↪subdeviceId, double FreqMHz)
    uint32_t numFreqDomains
    if ((zetSysmanFrequencyGet(hSysmanDevice, &numFreqDomains, NULL) == ZE_RESULT_
↪SUCCESS))
        zet_sysman_freq_handle_t* pFreqHandles =
            allocate_memory(numFreqDomains * sizeof(zet_sysman_freq_handle_t))
        if (zetSysmanFrequencyGet(hSysmanDevice, &numFreqDomains, pFreqHandles) == ZE_
↪RESULT_SUCCESS)
            for (index = 0 .. numFreqDomains-1)
                zet_freq_properties_t props
                if (zetSysmanFrequencyGetProperties(pFreqHandles[index], &props) ==␣
↪ZE_RESULT_SUCCESS)
                    # Only change the frequency of the domain if:
                    # 1. The domain controls a GPU accelerator
                    # 2. The domain frequency can be changed
                    # 3. The domain is located in the specified sub-device
                    if (props.type == ZET_FREQ_DOMAIN_GPU
                        and props.canControl
                        and props.subdeviceId == subdeviceId)
                        # Fix the frequency
                        zet_freq_range_t range
                        range.min = FreqMHz
                        range.max = FreqMHz
                        zetSysmanFrequencySetRange(pFreqHandles[index], &range)
    free_memory(...)
```

9.6 Events

Events are a way to determine if changes have occurred on a device e.g. new RAS errors without polling the Sysman API. An application registers the events that it wishes to receive notification about and then it listens for notifications. The application can choose to block when listening - this will put the calling application thread to sleep until new notifications are received.

The API enables registering for events from multiple devices and listening for any events coming from any devices by using one function call.

Once notifications have occurred, the application can use the query Sysman interface functions to get more details.

The following events are provided:

- Any RAS errors have occurred

The full list of available functions for handling events is described *below*.

CHAPTER TEN

INTERFACE DETAILS

10.1 Global operations

10.1.1 Device properties

The following operations permit getting properties about the entire device:

Function	Description
::zetSysmanDeviceGetProperties()	Get static device properties - device UUID, sub-device ID, device brand/model/vendor strings
::zetSysmanDeviceGetRepairStatus()	Determine if the device has undergone repairs, either through the running of diagnostics or by manufacturing.

The pseudo code below shows how to display general information about a device:

```
function ShowDeviceInfo(zet_sysman_handle_t hSysmanDevice)
    zet_sysman_properties_t devProps
    zet_repair_status_t repaired
    if (zetSysmanDeviceGetProperties(hSysmanDevice, &devProps) == ZE_RESULT_SUCCESS)
        output("    UUID:           %s", devProps.core.uuid.id)
        output("    #subdevices:    %u", devProps.numSubdevices)
        output("    brand:          %s", devProps.brandName)
        output("    model:          %s", devProps.modelName)
    if (zetSysmanDeviceGetRepairStatus(hSysmanDevice, &repaired) == ZE_RESULT_SUCCESS)
        output("    Was repaired:   %s", (repaired == ZET_REPAIR_STATUS_PERFORMED) ?
↪"yes" : "no")
```

10.1.2 Host processes

The following functions provide information about host processes that are using the device:

Function	Description
::zetSysmanProcessesGetState()	Get information about all processes that are using this device - process ID, device memory allocation size, accelerators being used.

Using the process ID, an application can determine the owner and the path to the executable - this information is not returned by the API.

10.1.3 Scheduler operations

On some devices, it is possible to change the way the scheduler executes workloads. To find out if this is supported, execute the function ::zetSysmanSchedulerGetCurrentMode() and check that it does not return an error.

The available scheduler operating modes are given by the enum ::zet_sched_mode_t:

Scheduler mode	Description
::ZET_SCHED_MODE_TIMEOUT	This mode is optimized for multiple applications or contexts submitting work to the hardware. When higher priority work arrives, the scheduler attempts to pause the current executing work within some timeout interval, then submits the other work.It is possible to configure (::zet_sched_timeout_properties_t) the watchdog timeout which controls the maximum time the scheduler will wait for a workload to complete a batch of work or yield to other applications before it is terminated.If the watchdog timeout is set to ::ZET_SCHED_WATCHDOG_DISABLE, the scheduler enforces no fairness. This means that if there is other work to execute, the scheduler will try to submit it but will not terminate an executing process that does not complete quickly.
::ZET_SCHED_MODE_TIMESLICE	This mode is optimized to provide fair sharing of hardware execution time between multiple contexts submitting work to the hardware concurrently.It is possible to configure (::zet_sched_timeslice_properties_t) the timeslice interval and the amount of time the scheduler will wait for work to yield to another application before it is terminated.
::ZET_SCHED_MODE_EXCLUSIVE	This mode is optimized for single application/context use-cases. It permits a context to run indefinitely on the hardware without being preempted or terminated. All pending work for other contexts must wait until the running context completes with no further submitted work.
::ZET_SCHED_MODE_COMPUTE_UNIT_DEBUG	This mode is optimized for application debug. It ensures that only one command queue can execute work on the hardware at a given time. Work is permitted to run as long as needed without enforcing any scheduler fairness policies.

The following functions are available for changing the behavior of the scheduler:

Function	Description
::zetSysmanSchedulerGetCurrentMode()	Get the current scheduler mode (timeout, timeslice, exclusive, single command queue)
::zetSysmanSchedulerGetTimeoutModeProperties()	Get the settings for the timeout scheduler mode
::zetSysmanSchedulerGetTimesliceModeProperties()	Get the settings for the timeslice scheduler mode
::zetSysmanSchedulerSetTimeoutMode()	Change to timeout scheduler mode and/or change properties
::zetSysmanSchedulerSetTimesliceMode()	Change to timeslice scheduler mode and/or change properties
::zetSysmanSchedulerSetExclusiveMode()	Change to exclusive scheduler mode and/or change properties
::zetSysmanSchedulerSetComputeUnitDebugMode()	Change to compute unit debug scheduler mode and/or change properties

The pseudo code below shows how to stop the scheduler enforcing fairness while permitting other work to attempt to run:

```
function DisableSchedulerWatchdog(zet_sysman_handle_t hSysmanDevice)
    ze_result_t res
```

(continues on next page)

(continued from previous page)

```
    zet_sched_mode_t currentMode
    res = zetSysmanSchedulerGetCurrentMode(hSysmanDevice, &currentMode)
    if (res == ZE_RESULT_SUCCESS)
        ze_bool_t requireReboot
        zet_sched_timeout_properties_t props
        props.watchdogTimeout = ZET_SCHED_WATCHDOG_DISABLE
        res = zetSysmanSchedulerSetTimeoutMode(hSysmanDevice, &props, &requireReboot)
        if (res == ZE_RESULT_SUCCESS)
            if (requireReboot)
                output("WARNING: Reboot required to complete desired configuration.")
            else
                output("Schedule mode changed successfully.")
        else if(res == ZE_RESULT_ERROR_UNSUPPORTED_FEATURE)
            output("ERROR: The timeout scheduler mode is not supported on this␣
↪device.")
        else if(res == ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS)
            output("ERROR: Don't have permissions to change the scheduler mode.")
        else
            output("ERROR: Problem calling the API to change the scheduler mode.")
    else if(res == ZE_RESULT_ERROR_UNSUPPORTED_FEATURE)
        output("ERROR: Scheduler modes are not supported on this device.")
    else
        output("ERROR: Problem calling the API.")
```

10.1.4 Device reset

The device can be reset using the following function:

Function	Description
::zetSysmanDeviceReset()	Requests that the driver reset the device. If the hardware is hung, this will perform an PCI bus reset.

10.1.5 PCI link operations

The following functions permit getting data about the PCI endpoint for the device:

Function	Description
::zetSysmanPciGetProperties()	Get static properties for the PCI port - BDF address, number of bars, maximum supported speed
::zetSysmanPciGetState()	Get current PCI port speed (number of lanes, generation)
::zetSysmanPciGetBars()	Get information about each configured PCI bar
::zetSysmanPciGetStats()	Get PCI statistics - throughput, total packets, number of packet replays

The pseudo code below shows how to output the PCI BDF address:

```
function ShowPciInfo(zet_sysman_handle_t hSysmanDevice)
    zet_pci_properties_t pciProps;
    if (zetSysmanPciGetProperties(hSysmanDevice, &pciProps) == ZE_RESULT_SUCCESS)
        output("    PCI address:        %04u:%02u:%02u.%u",
            pciProps.address.domain,
            pciProps.address.bus,
```

(continues on next page)

(continued from previous page)

```
            pciProps.address.device,
            pciProps.address.function);
```

10.2 Operations on power domains

The PSU (Power Supply Unit) provides power to a device. The amount of power drawn by a device is a function of the voltage and frequency, both of which are controlled by the Punit, a micro-controller on the device. If the voltage and frequency are too high, two conditions can occur:

1. Over-current - This is where the current drawn by the device exceeds the maximum current that the PSU can supply. The PSU asserts a signal when this occurs, and it is processed by the Punit.
2. Over-temperature - The device is generating too much heat that cannot be dissipated fast enough. The Punit monitors temperatures and reacts when the sensors show the maximum temperature exceeds the threshold TjMax (typically 100 degrees Celsius).

When either of these conditions occurs, the Punit throttles the frequencies/voltages of the device down to their minimum values, severely impacting performance. The Punit avoids such severe throttling by measuring the actual power being consumed by the system and slowly throttling the frequencies down when power exceeds some limits. Three limits are monitored by the Punit:

Limit	Window	Description
Peak	Instantaneous	Punit tracks the instantaneous power. When this exceeds a programmable threshold, the Punit will aggressively throttle frequencies/voltages. The threshold is referred to as PL4 - Power Limit 4 - or peak power.
Burst	2ms	Punit tracks the 2ms moving average of power. When this exceeds a programmable threshold, the Punit starts throttling frequencies/voltages. The threshold is referred to as PL2 - Power Limit 2 - or burst power.
Sustained	28sec	Punit tracks the 28sec moving average of power. When this exceeds a programmable threshold, the Punit throttles frequencies/voltages. The threshold is referred to as PL1 - Power Limit 1 - or sustained power.

Peak power limit is generally greater than the burst power limit which is generally greater than the sustained power limit. The default factory values are tuned assuming the device is operating at normal temperatures running significant workloads:

- The peak power limit is tuned to avoid tripping the PSU over-current signal for all but the most intensive compute workloads. Most workloads should be able to run at maximum frequencies without hitting this condition.
- The burst power limit permits most workloads to run at maximum frequencies for short periods.
- The sustained power limit will be triggered if high frequencies are requested for lengthy periods (configurable, default is 28sec) and the frequencies will be throttled if the high requests and utilization of the device continues.

Some power domains support requesting the event ::ZET_SYSMAN_EVENT_TYPE_ENERGY_THRESHOLD_CROSSED be generated when the energy consumption exceeds some value. This can be a useful technique to suspend an application until the GPU becomes busy. The technique involves calling ::zetSysmanPowerSetEnergyThreshold() with some delta energy threshold, registering to receive the event using the function ::zetSysmanEventSetConfig() and then calling ::zetSysmanEventListen() to block until the event is triggered. When the energy consumed by the power domain from the time the call is made exceeds the specified delta, the event is triggered, and the application is woken up.

The following functions are provided to manage the power of the device:

Function	Description
::zetSysmanPowerGet()	Enumerate the power domains.
::zetSysmanPowerGetProperties()	Get the maximum power limit that can be specified when changing the power limits of a specific power domain.
::zetSysmanPowerGetEnergyCounter()	Read the energy consumption of the specific domain.
::zetSysmanPowerGetLimits()	Get the sustained/burst/peak power limits for the specific power domain.
::zetSysmanPowerSetLimits()	Set the sustained/burst/peak power limits for the specific power domain.
::zetSysmanPowerGetEnergyThreshold()	Get the current energy threshold.
::zetSysmanPowerSetEnergyThreshold()	Set the energy threshold. Event ::ZET_SYSMAN_EVENT_TYPE_ENERGY_THRESHOLD_CROSSED will be generated when the energy consumed since calling this function exceeds the specified threshold.

The pseudo code below shows how to output information about each power domain on a device:

```
function ShowPowerDomains(zet_sysman_handle_t hSysmanDevice)
    uint32_t numPowerDomains
    if (zetSysmanPowerGet(hSysmanDevice, &numPowerDomains, NULL) == ZE_RESULT_SUCCESS)
        zet_sysman_pwr_handle_t* phPower =
            allocate_memory(numPowerDomains * sizeof(zet_sysman_pwr_handle_t))
        if (zetSysmanPowerGet(hSysmanDevice, &numPowerDomains, phPower) == ZE_RESULT_
↪SUCCESS)
            for (pwrIndex = 0 .. numPowerDomains-1)
                zet_power_properties_t props
                if (zetSysmanPowerGetProperties(phPower[pwrIndex], &props) == ZE_
↪RESULT_SUCCESS)
                    if (props.onSubdevice)
                        output("Sub-device %u power:\n", props.subdeviceId)
                        output("    Can control: %s", props.canControl ? "yes" : "no")
                        call_function ShowPowerLimits(phPower[pwrIndex])
                    else
                        output("Total package power:\n")
                        output("    Can control: %s", props.canControl ? "yes" : "no")
                        call_function ShowPowerLimits(phPower[pwrIndex])
    free_memory(...)
}

function ShowPowerLimits(zet_sysman_pwr_handle_t hPower)
    zet_power_sustained_limit_t sustainedLimits
    zet_power_burst_limit_t burstLimits
    zet_power_peak_limit_t peakLimits
    if (zetSysmanPowerGetLimits(hPower, &sustainedLimits, &burstLimits, &peakLimits)␣
↪== ZE_RESULT_SUCCESS)
        output("    Power limits\n")
        if (sustainedLimits.enabled)
            output("        Sustained: %.3f W %.3f sec",
                sustainedLimits.power / 1000,
```

(continues on next page)

(continued from previous page)

```
                sustainedLimits.interval / 1000)
        else
            output("        Sustained: Disabled")
        if (burstLimits.enabled)
            output("        Burst:     %.3f", burstLimits.power / 1000)
        else
            output("        Burst:     Disabled")
        output("        Burst:     %.3f", peakLimits.power / 1000)
```

The pseudo code shows how to output the average power. It assumes that the function is called regularly (say every 100ms).

```
function ShowAveragePower(zet_sysman_pwr_handle_t hPower, zet_power_energy_counter_t*␣
↪pPrevEnergyCounter)
    zet_power_energy_counter_t newEnergyCounter;
    if (zetSysmanPowerGetEnergyCounter(hPower, &newEnergyCounter) == ZE_RESULT_
↪SUCCESS)
        uint64_t deltaTime = newEnergyCounter.timestamp - pPrevEnergyCounter->
↪timestamp;
        if (deltaTime)
            output("    Average power: %.3f W",
                (newEnergyCounter.energy - pPrevEnergyCounter->energy) / deltaTime);
            *pPrevEnergyCounter = newEnergyCounter;
```

10.3 Operations on frequency domains

The hardware manages frequencies to achieve a balance between best performance and power consumption. Most devices have one or more frequency domains.

The following functions are provided to manage the frequency domains on the device:

Function	Description
::zetSysmanFrequencyGet()	Enumerate all the frequency domains on the device and sub-devices.
::zetSysmanFrequencyGet-Properties()	Find out which domain ::zet_freq_domain_t is controlled by this frequency and min/max hardware frequencies.
::zetSysmanFrequencyGe-tAvailableClocks()	Get an array of all available frequencies that can be requested on this domain.
::zetSysmanFrequencyGe-tRange()	Get the current min/max frequency between which the hardware can operate for a frequency domain.
::zetSysmanFrequencySe-tRange()	Set the min/max frequency between which the hardware can operate for a fre-quency domain.
::zetSysmanFrequencyGet-State()	Get the current frequency request, actual frequency, TDP frequency and throttle reasons for a frequency domain.
::zetSysmanFrequencyGet-ThrottleTime()	Gets the amount of time a frequency domain has been throttled.

It is only permitted to set the frequency range if the device property ::zet_freq_properties_t.canControl is true for the specific frequency domain.

By setting the min/max frequency range to the same value, software is effectively disabling the hardware-controlled frequency and getting a fixed stable frequency providing the Punit does not need to throttle due to excess power/heat.

Based on the power/thermal conditions, the frequency requested by software or the hardware may not be respected. This situation can be determined using the function ::zetSysmanFrequencyGetState() which will indicate the current frequency request, the actual (resolved) frequency and other frequency information that depends on the current conditions. If the actual frequency is below the requested frequency, ::zet_freq_state_t.throttleReasons will provide the reasons why the frequency is being limited by the Punit.

When a frequency domain starts being throttled, the event ::ZET_SYSMAN_EVENT_TYPE_FREQ_THROTTLED is triggered if this is supported (check ::zet_freq_properties_t.isThrottleEventSupported).

10.3.1 Frequency/Voltage overclocking

Overclocking involves modifying the voltage-frequency (V-F) curve to either achieve better performance by permitting the hardware to reach higher frequencies or better efficiency by lowering the voltage for the same frequency.

By default, the hardware imposes a factory-fused maximum frequency and a voltage-frequency curve. The voltage-frequency curve specifies how much voltage is needed to safely reach a given frequency without hitting overcurrent conditions. If the hardware detects overcurrent (IccMax), it will severely throttle frequencies in order to protect itself. Also, if the hardware detects that any part of the chip exceeds a maximum temperature limit (TjMax) it will also severely throttle frequencies.

To improve maximum performance, the following modifications can be made:

- Increase the maximum frequency.
- Increase the voltage to ensure stability at the higher frequency.
- Increase the maximum current (IccMax).
- Increase the maximum temperature (TjMax).

All these changes come with the risk of damage the device.

To improve efficiency for a given workload that is not excercising the full circuitry of the device, the following modifications can be made:

- Decrease the voltage

Frequency/voltage overclocking is accomplished by calling ::zetSysmanFrequencyOcSetConfig() with a new overclock configuration ::zet_oc_config_t. There are two modes that control the way voltage is handled when overclocking the frequency:

Voltage overclock mode	Description
::ZET_OC_MODE_OVERRIDE	In this mode, a fixed user-supplied voltage (::zet_oc_config_t.voltageTarget + ::zet_oc_config_t.voltageOffset) is applied at all times, independent of the frequency request. This is not efficient but can improve stability by avoiding power-supply voltage changes as the frequency changes.
::ZET_OC_MODE_OVERRIDE	In this mode, a fixed user-supplied voltage is applied at all times, independent of the frequency request. This is not efficient but can improve stability by avoiding power-supply voltage changes as the frequency changes. Generally, this mode is used in conjunction with a fixed frequency.

The following functions are provided to handle overclocking:

Function	Description
::zetSysmanFrequencyOcGetCapabilities()	Determine the overclock capabilities of the device.
::zetSysmanFrequencyOcGetConfig()	Get the overclock configuration in effect.
::zetSysmanFrequencyOcSetConfig()	Set a new overclock configuration.
::zetSysmanFrequencyOcGetIccMax()	Get the maximum current limit in effect.
::zetSysmanFrequencyOcSetIccMax()	Set a new maximum current limit.
::zetSysmanFrequencyOcGetTjMax()	Get the maximum temperature limit in effect.
::zetSysmanFrequencyOcSetTjMax()	Set a new maximum temperature limit.

Overclocking can be turned off by calling ::zetSysmanFrequencyOcSetConfig() with mode ::ZET_OC_MODE_OFF and by calling zetSysmanFrequencyOcGetIccMax() and ::zetSysmanFrequencyOcSetTjMax() with values of 0.0.

10.4 Operations on engine groups

It is possible to monitor the activity of one or engines combined into an **engine group**. A device can have multiple engine groups and the possible types are defined in ::zet_engine_group_t. The current engine groups supported are global activity across all engines, activity across all compute accelerators, activity across all media accelerators and activity across all copy engines.

By taking two snapshots of the activity counters, it is possible to calculate the average utilization of different parts of the device.

The following functions are provided:

Function	Description
::zetSysmanEngineGet()	Enumerate the engine groups that can be queried.
::zetSysmanEngineGetProperties()	Get the properties of an engine group. This will return the type of engine group (one of ::zet_engine_group_t) and on which sub-device the group is making measurements.
::zetSysmanEngineGetActivity()	Returns the activity counters for an engine group.

10.5 Operations on standby domains

When a device is idle, it will enter a low-power state. Since exit from low-power states have associated latency, it can hurt performance. The hardware attempts to stike a balance between saving power when there are large idle times between workload submissions to the device and keeping the device awake when it determines that the idle time between submissions is short.

A device can consist of one or more standby domains - the list of domains is given by ::zet_standby_type_t.

The following functions can be used to control how the hardware promotes to standby states:

Function	Description
::zetSysman-StandbyGet()	Enumerate the standby domains.
::zetSysman-StandbyGetProp-erties()	Get the properties of a standby domain. This will return the parts of the device that are affected by this domain (one of ::zet_engine_group_t) and on which sub-device the domain is located.
::zetSysman-StandbyGet-Mode()	Get the current promotion mode (one of ::zet_standby_promo_mode_t) for a standby domain.
::zetSysman-StandbySet-Mode()	Set the promotion mode (one of ::zet_standby_promo_mode_t) for a standby domain.

10.6 Operations on firmwares

The following functions are provided to manage firmwares on the device:

Function	Description
::zetSysmanFirmwareGet()	Enumerate all firmwares that can be managed on the device.
::zetSysmanFirmwareGetProperties()	Find out the name and version of a firmware.
::zetSysmanFirmwareGetChecksum()	Get the checksum for an installed firmware.
::zetSysmanFirmwareFlash()	Flash a new firmware image.

10.7 Querying memory modules

The API provides an enumeration of all device memory modules. For each memory module, the current and maximum bandwidth can be queried. The API also provides a health metric which can take one of the following values (::zet_mem_health_t):

Memory health	Description
::ZET_MEM_HEALTH_OK	All memory channels are healthy.
::ZET_MEM_HEALTH_DEGRADED	Excessive correctable errors have been detected on one or more channels. Device should be reset.
::ZET_MEM_HEALTH_CRITICAL	Operating with reduced memory to cover banks with too many uncorrectable errors.
::ZET_MEM_HEALTH_REPLACE	Device should be replaced due to excessive uncorrectable errors.

When the health state of a memory module changes, the event ::ZET_SYSMAN_EVENT_TYPE_MEM_HEALTH is triggered.

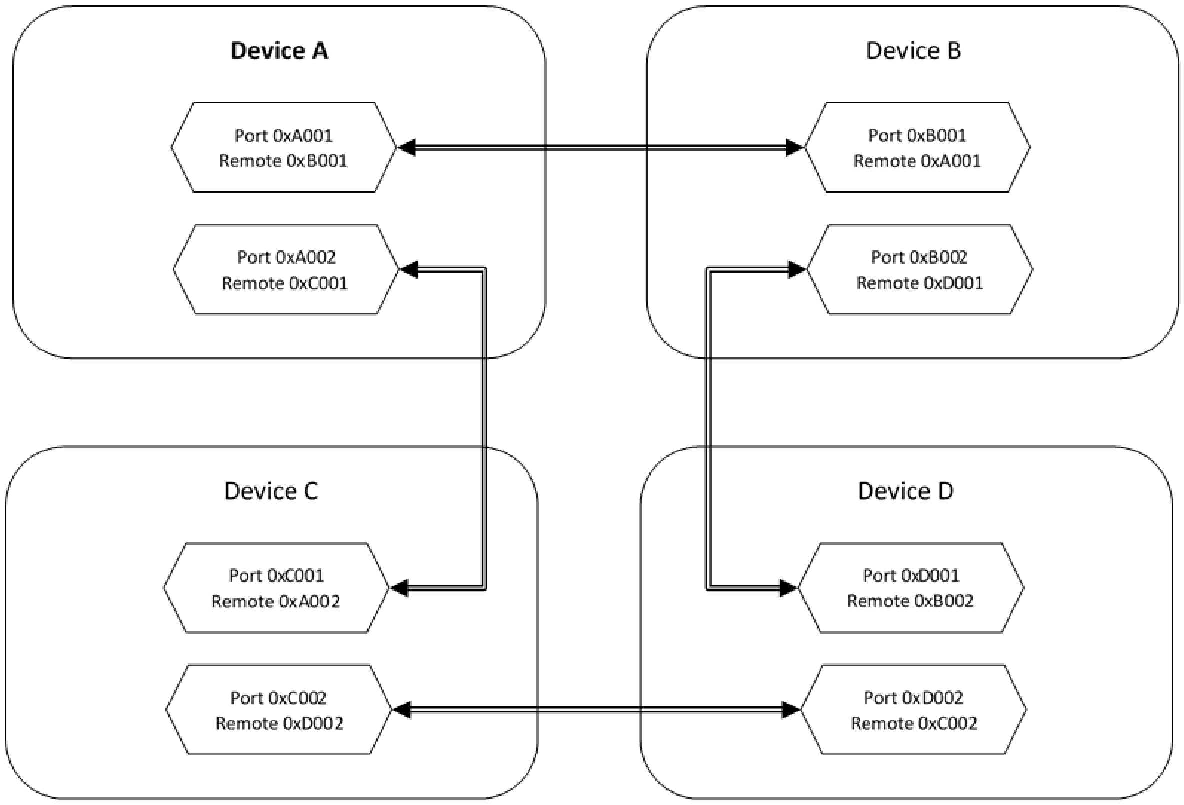
The following functions provide access to information about the device memory modules:

Function	Description
::zetSysmanMemoryGet()	Enumerate the memory modules.
::zetSysmanMemoryGetProperties()	Find out the type of memory and maximum physical memory of a module.
::zetSysmanMemoryGetBand-width()	Returns memory bandwidth counters for a module.
::zetSysmanMemoryGetState()	Returns the currently health and allocated memory size for a module.

10.8 Operations on Fabric ports

Fabric is the term given to describe high-speed interconnections between accelerator devices, primarily used to provide low latency fast access to remote device memory. Devices have one or more **fabric ports** that transmit and receive data over physical links. Links connect fabric ports, thus permitting data to travel between devices. Routing rules determine the flow of traffic through the fabric.

The figure below shows four devices, each with two fabric ports. Each port has a link that connects it to a port on another device. In this example, the devices are connected in a ring. Device A and D can access each other's memory through either device B or device C depending on how the fabric routing rules are configured. If the connection between device B and D goes down, the routing rules can be modified such that device B and D can still access each other's memory by going through two hops in the fabric (device A and C).



The API permits enumerating all the ports available on a device. Each port has a universal unique identifier (UUID). If the port is connected to another port, the API will provide the remote port's UUID. By enumerating all ports on all devices that are connected to the fabric, an application can build a topology map of connectivity.

For each port, the API permits querying its configuration (UP/DOWN) and its health which can take one of the following values:

Fabric port health	Description
::ZET_FABRIC_PORT_STATUS_GREEN	The port is up and operating as expected.
::ZET_FABRIC_PORT_STATUS_YELLOW	The port is up but has quality and/or bandwidth degradation.
::ZET_FABRIC_PORT_STATUS_RED	Port connection instabilities are preventing workloads making forward progress.
::ZET_FABRIC_PORT_STATUS_BLACK	The port is configured down.

If the port is in a yellow state, the API provides additional information about the types of quality degradation that are being observed. If the port is in a red state, the API provides additional information about the causes of the instability.

When a port's health state changes, the event ::ZET_SYSMAN_EVENT_TYPE_FABRIC_PORT_HEALTH is triggered.

The API permits measuring the receive and transmit bandwidth flowing through each port. It also provides the maximum receive and transmit speed (frequency/number of lanes) of each port and the current speeds which can be lower if operating in a degraded state. Note that a port's receive and transmit speeds are not necessarily the same.

Since ports can pass data directly through to another port, the measured bandwidth at a port can be higher than the actual bandwidth generated by the accelerators directly connected by two ports. As such, bandwidth metrics at each port are more relevant for determining points of congestion in the fabric and less relevant for measuring the total bandwidth passing between two accelerators.

The following functions can be used to manage Fabric ports:

Function	Description
::zetSysmanFabricPort-Get()	Enumerate all fabric ports on the device.
::zetSysmanFabricPort-GetProperties()	Get static properties about the port (model, UUID, max receive/transmit speed).
::zetSysmanFabricPort-GetLinkType()	Get details about the physical link connected to the port.
::zetSysmanFabricPort-GetConfig()	Determine if the port is configured UP and if beaconing is on or off.
::zetSysmanFabricPort-SetConfig()	Configure the port UP or DOWN and turn beaconing on or off.
::zetSysmanFabricPort-GetState()	Determine the health of the port connection, reasons for link degradation or connection issues and the current receive/transmit speed.
::zetSysmanFabricPort-GetThroughput()	Get port receive/transmit counters along with current receive/transmit port speed.

For devices with sub-devices, the fabric ports are usually located in the sub-device. Given a device handle, ::zetSysmanFabricPortGet() will include the ports on each sub-device. In this case, ::zet_fabric_port_properties_t.onSubdevice will be set to true and ::zet_fabric_port_properties_t.subdeviceId will give the subdevice ID where that port is located.

The pseudo-code below shows how to get the state of all fabric ports in the device and sub-devices:

```
void ShowFabricPorts(zet_sysman_handle_t hSysmanDevice)
    uint32_t numPorts
    if ((zetSysmanFabricPortGet(hSysmanDevice, &numPorts, NULL) == ZE_RESULT_SUCCESS))
        zet_sysman_fabric_port_handle_t* phPorts =
            allocate_memory(numPorts * sizeof(zet_sysman_fabric_port_handle_t))
        if (zetSysmanFabricPortGet(hSysmanDevice, &numPorts, phPorts) == ZE_RESULT_
↪SUCCESS)
            for (index = 0 .. numPorts-1)
                # Show information about a particular port
                output("    Port %u:\n", index)
                call_function ShowFabricPortInfo(phPorts[index])
    free_memory(...)

function ShowFabricPortInfo(zet_sysman_fabric_port_handle_t hPort)
    zet_fabric_port_properties_t props
    if (zetSysmanFabricPortGetProperties(hPort, &props) == ZE_RESULT_SUCCESS)
        zet_fabric_port_state_t state
        if (zetSysmanFabricPortGetState(hPort, &state) == ZE_RESULT_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
            zet_fabric_link_type_t link
            if (zetSysmanFabricPortGetLinkType(hPort, false, &link) == ZE_RESULT_
↪SUCCESS)
                zet_fabric_port_config_t config
                if (zetSysmanFabricPortGetConfig(hPort, &config) == ZE_RESULT_SUCCESS)
                    output("        Model:                 %s", props.model)
                    if (props.onSubdevice)
                        output("        On sub-device:         %u", props.subdeviceId)
                    if (config.enabled)
                    {
                        var status
                        output("        Config:                UP")
                        switch (state.status)
                            case ZET_FABRIC_PORT_STATUS_GREEN:
                                status = "GREEN - The port is up and operating as␣
↪expected"
                            case ZET_FABRIC_PORT_STATUS_YELLOW:
                                status = "YELLOW - The port is up but has quality and/
↪or bandwidth degradation"
                            case ZET_FABRIC_PORT_STATUS_RED:
                                status = "RED - Port connection instabilities"
                            case ZET_FABRIC_PORT_STATUS_BLACK:
                                status = "BLACK - The port is configured down"
                            default:
                                status = "UNKNOWN"
                        output("        Status:                %s", status)
                        output("        Link type:             %s", link.desc)
                        output(
                            "        Max speed (rx/tx):     %llu/%llu bytes/sec",
                            props.maxRxSpeed.maxBandwidth,
                            props.maxTxSpeed.maxBandwidth)
                        output(
                            "        Current speed (rx/tx): %llu/%llu bytes/sec",
                            state.rxSpeed.maxBandwidth,
                            state.txSpeed.maxBandwidth)
                    else
                        output("        Config:                DOWN")
```

10.9 Querying temperature

A device has multiple temperature sensors embedded at different locations. The following locations are supported:

Temperature sensor location	Description
::ZET_TEMP_SENSORS_GLOBAL	Returns the maximum measured across all sensors in the device.
::ZET_TEMP_SENSORS_GPU	Returns the maximum measured across all sensors in the GPU accelerator.
::ZET_TEMP_SENSORS_MEMORY	Returns the maximum measured across all sensors in the device memory.

For some sensors, it is possible to request that events be triggered when temperatures cross thresholds. This is accomplished using the function ::zetSysmanTemperatureGetConfig() and ::zetSysmanTemperatureSetConfig(). Support for specific events is accomplished by calling ::zetSysmanTemperatureGetProperties(). In general, temperature events are only supported on the temperature sensor of type ::ZET_TEMP_SENSORS_GLOBAL. The list below describes the list of temperature events:

Event	Check support	Description
::ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL	::zet_temp_properties_t.isCriticalTempSupported	The event is triggered when the temperature crosses into the critical zone where severe frequency throttling will be taking place.
::ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1	::zet_temp_properties_t.isThreshold1Supported	The event is triggered when the temperature crosses the custom threshold 1. Flags can be set to limit the trigger to when crossing from high to low or low to high.
::ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD2 PE_TEMP_THRESHOLD2	::zet_temp_properties_t.isThreshold2Supported	The event is triggered when the temperature crosses the custom threshold 2. Flags can be set to limit the trigger to when crossing from high to low or low to high.

The following function can be used to manage temperature sensors:

Function	Description
::zetSysmanTemperatureGet()	Enumerate the temperature sensors on the device.
::zetSysmanTemperatureGetProperties()	Get static properties for a temperature sensor. In particular, this will indicate which parts of the device the sensor measures (one of ::zet_temp_sensors_t).
::zetSysmanTemperatureGetConfig()	Get information about the current temperature thresholds - enabled/threshold/processID.
::zetSysmanTemperatureSetConfig()	Set new temperature thresholds. Events will be triggered when the temperature crosses these thresholds.
::zetSysmanTemperatureGetState()	Read the temperature of a sensor.

10.10 Operations on power supplies

The following functions can be used to access information about each power-supply on a device:

Function	Description
::zetSysmanPsuGet()	Enumerate the power supplies on the device that can be managed.
::zetSysmanPsuGetProperties()	Get static details about the power supply.
::zetSysmanPsuGetState()	Get information about the health (temperature, current, fan) of the power supply.

10.11 Operations on fans

If ::zetSysmanFanGet() returns one or more fan handles, it is possible to manage their speed. The hardware can be instructed to run the fan at a fixed speed (or 0 for silent operations) or to provide a table of temperature-speed points in which case the hardware will dynamically change the fan speed based on the current temperature of the chip. This configuration information is described in the structure ::zet_fan_config_t. When specifying speed, one can provide the value in revolutions per minute (::ZET_FAN_SPEED_UNITS_RPM) or as a percentage of the maximum RPM (::ZET_FAN_SPEED_UNITS_PERCENT).

The following functions are available:

Function	Description
::zetSysmanFanGet()	Enumerate the fans on the device.
::zetSysmanFanGet-Properties()	Get the maximum RPM of the fan and the maximum number of points that can be specified in the temperature-speed table for a fan.
::zetSysmanFanGet-Config()	Get the current configuration (speed) of a fan.
::zetSysmanFanSet-Config()	Change the configuration (speed) of a fan.
::zetSysmanFanGet-State()	Get the current speed of a fan.

The pseudo code below shows how to output the fan speed of all fans:

```
function ShowFans(zet_sysman_handle_t hSysmanDevice)
    uint32_t numFans
    if (zetSysmanFanGet(hSysmanDevice, &numFans, NULL) == ZE_RESULT_SUCCESS)
        zet_sysman_fan_handle_t* phFans =
            allocate_memory(numFans * sizeof(zet_sysman_fan_handle_t))
        if (zetSysmanFanGet(hSysmanDevice, &numFans, phFans) == ZE_RESULT_SUCCESS)
            output("    Fans")
            for (fanIndex = 0 .. numFans-1)
                uint32_t speed
                if (zetSysmanFanGetState(phFans[fanIndex], ZET_FAN_SPEED_UNITS_RPM, &
↪speed)
                    == ZE_RESULT_SUCCESS)
                        output("        Fan %u: %u RPM", fanIndex, speed)
    free_memory(...)
}
```

The next example shows how to set the fan speed for all fans to a fixed value in RPM, but only if control is permitted:

```
function SetFanSpeed(zet_sysman_handle_t hSysmanDevice, uint32_t SpeedRpm)
{
    uint32_t numFans
    if (zetSysmanFanGet(hSysmanDevice, &numFans, NULL) == ZE_RESULT_SUCCESS)
        zet_sysman_fan_handle_t* phFans =
            allocate_memory(numFans * sizeof(zet_sysman_fan_handle_t))
        if (zetSysmanFanGet(hSysmanDevice, &numFans, phFans) == ZE_RESULT_SUCCESS)
            zet_fan_config_t config
            config.mode = ZET_FAN_SPEED_MODE_FIXED
            config.speed = SpeedRpm
            config.speedUnits = ZET_FAN_SPEED_UNITS_RPM
            for (fanIndex = 0 .. numFans-1)
                zet_fan_properties_t fanprops
                if (zetSysmanFanGetProperties(phFans[fanIndex], &fanprops) == ZE_
↪RESULT_SUCCESS)
                    if (fanprops.canControl)
                        zetSysmanFanSetConfig(phFans[fanIndex], &config)
                    else
                        output("ERROR: Can't control fan %u.\n", fanIndex)
    free_memory(...)
}
```

10.12 Operations on LEDs

If ::zetSysmanLedGet() returns one or more LED handles, it is possible to manage LEDs on the device. This includes turning them off/on and where the capability exists, changing their color in real-time.

The following functions are available:

Function	Description
::zetSysmanLedGet()	Enumerate the LEDs on the device that can be managed.
::zetSysmanLedGetProperties()	Find out if a LED supports color changes.
::zetSysmanLedGetState()	Find out if a LED is currently off/on and the color where the capability is available.
::zetSysmanLedSetState()	Turn a LED off/on and set the color where the capability is available.

10.13 Querying RAS errors

RAS stands for Reliability, Availability and Serviceability. It is a feature of certain devices that attempts to correct random bit errors and provide redundancy where permanent damage has occurred.

If a device supports RAS, it maintains counters for hardware and software errors. There are two types of errors and they are defined in ::zet_ras_error_type_t:

Error Type	Description
::ZET_RAS_ERROR_TYPE_UNCORRECTABLE	Hardware errors occurred which most likely resulted in loss of data or even a device hang. If an error results in device lockup, a warm boot is required before those errors will be reported.
::ZET_RAS_ERROR_TYPE_CORRECTABLE	These are errors that were corrected by the hardware and did not cause data corruption.

Software can use the function ::zetSysmanRasGetProperties() to find out if the device supports RAS and if it is enabled. This information is returned in the structure ::zet_ras_properties_t.

The function ::zetSysmanRasGet() enumerates the available sets of RAS errors. If no handles are returned, the device does not support RAS. A device without sub-devices will return one handle if RAS is supported. A device with sub-devices will return a handle for each sub-device.

To determine if errors have occurred, software uses the function ::zetSysmanRasGetState(). This will return the total number of errors of a given type (correctable/uncorrectable) that have occurred.

When calling ::zetSysmanRasGetState(), software can request that the error counters be cleared. When this is done, all counters of the specified type (correctable/uncorrectable) will be set to zero and any subsequent calls to this function will only show new errors that have occurred. If software intends to clear errors, it should be the only application doing so and it should store the counters in an appropriate database for historical analysis.

When calling ::zetSysmanRasGetState(), an optional pointer to a structure of type ::zet_ras_details_t can be supplied. This will give a breakdown of the main device components where the errors occurred. The categories are defined in the structure ::zet_ras_details_t. The meaning of each category depends on the error type (correctable, uncorrectable).

Error category	::ZET_RAS_ERROR_TYPE_CORRECTABLE	::ZET_RAS_ERROR_TYPE_UNCORRECTABLE
::zet_ras_details_t.numResets	Always zero.	Number of device resets that have taken place.
::zet_ras_details_t.numProgrammingErrors	Always zero.	Number of hardware exceptions generated by the way workloads have programmed the hardware.
::zet_ras_details_t.numDriverErrors	Always zero.	Number of low level driver communication errors have occurred.
::zet_ras_details_t.numComputeErrors	Number of errors that have occurred in the accelerator hardware that were corrected.	Number of errors that have occurred in the accelerator hardware that were not corrected. These would have caused the hardware to hang and the driver to reset.
::zet_ras_details_t.numNonComputeErrors	Number of errors occurring in fixed-function accelerator hardware that were corrected.	Number of errors occurring in the fixed-function accelerator hardware there could not be corrected. Typically these will result in a PCI bus reset and driver reset.
::zet_ras_details_t.numCacheErrors	Number of ECC correctable errors that have occurred in the on-chip caches (caches/register file/shared local memory).	Number of ECC uncorrectable errors that have occurred in the on-chip caches (caches/register file/shared local memory). These would have caused the hardware to hang and the driver to reset.
::zet_ras_details_t.numMemoryErrors	Number of times the device memory has transitioned from a healthy state to a degraded state. Degraded state occurs when the number of correctable errors cross a threshold.	Number of times the device memory has transitioned from a healthy/degraded state to a critical/replace state.
::zet_ras_details_t.numPciErrors:	controllerNumber of PCI packet replays that have occurred.	Number of PCI bus resets.
::zet_ras_details_t.numFabricErrors	Number of times one or more ports have transitioned from a green status to a yellow status. This indicates that links are experiencing quality degradation.	Number of times one or more ports have transitioned from a green/yellow status to a red status. This indicates that links are experiencing connectivity statibility issues.
::zet_ras_details_t.numDisplayErrors	Number of ECC correctable errors that have occurred in the display.	Number of ECC uncorrectable errors that have occurred in the display.

Each RAS error type can trigger events when the error counters exceed thresholds. The events are listed in the table below. Software can use the functions ::zetSysmanRasGetConfig() and ::zetSysmanRasSetConfig() to get and set the thresholds for each error type. The default is for all thresholds to be 0 which means that no events are generated. Thresholds can be set on the total RAS error counter or on each of the detailed error counters.

RAS error Type	Event
::ZET_RAS_ERROR_TYPE_UNCORRECTABLE	::ZET_SYSMAN_EVENT_TYPE_RAS_UNCORRECTABLE_ERRORS
::ZET_RAS_ERROR_TYPE_CORRECTABLE	::ZET_SYSMAN_EVENT_TYPE_RAS_CORRECTABLE_ERRORS

The table below summaries all the RAS management functions:

Function	Description
::zetSysmanRas-Get()	Get handles to the available RAS error groups.
::zetSysmanRasGet-Properties()	Get properties about a RAS error group - type of RAS errors and if they are enabled.
::zetSysmanRasGet-Config()	Get the current list of thresholds for each counter in the RAS group. RAS error events will be generated when the thresholds are exceeded.
::zetSysmanRasSet-Config()	Set current list of thresholds for each counter in the RAS group. RAS error events will be generated when the thresholds are exceeded.
::zetSysmanRasGet-State()	Get the current state of the RAS error counters. The counters can also be cleared.

The pseudo code below shows how to determine if RAS is supported and the current state of RAS errors:

```
void ShowRasErrors(zet_sysman_handle_t hSysmanDevice)
    uint32_t numRasErrorSets
    if ((zetSysmanRasGet(hSysmanDevice, &numRasErrorSets, NULL) == ZE_RESULT_SUCCESS))
        zet_sysman_ras_handle_t* phRasErrorSets =
            allocate_memory(numRasErrorSets * sizeof(zet_sysman_ras_handle_t))
        if (zetSysmanRasGet(hSysmanDevice, &numRasErrorSets, phRasErrorSets) == ZE_
↪RESULT_SUCCESS)
            for (rasIndex = 0 .. numRasErrorSets)
                zet_ras_properties_t props
                if (zetSysmanRasGetProperties(phRasErrorSets[rasIndex], &props) == ZE_
↪RESULT_SUCCESS)
                    var pErrorType
                    switch (props.type)
                        case ZET_RAS_ERROR_TYPE_CORRECTABLE:
                            pErrorType = "Correctable"
                        case ZET_RAS_ERROR_TYPE_UNCORRECTABLE:
                            pErrorType = "Uncorrectable"
                        default:
                            pErrorType = "Unknown"
                    output("RAS %s errors", pErrorType)
                    if (props.onSubdevice)
                        output("    On sub-device: %u", props.subdeviceId)
                    output("    RAS supported: %s", props.supported ? "yes" : "no")
                    output("    RAS enabled: %s", props.enabled ? "yes" : "no")
                    if (props.supported and props.enabled)
                        uint64_t newErrors
                        zet_ras_details_t errorDetails
                        if (zetSysmanRasGetState(phRasErrorSets[rasIndex], 1, &
↪newErrors, &errorDetails)
                            == ZE_RESULT_SUCCESS)
                                output("    Number new errors: %llu\n", (long long␣
↪unsigned int)newErrors)
                                if (newErrors)
                                    call_function OutputRasDetails(&errorDetails)
    free_memory(...)

function OutputRasDetails(zet_ras_details_t* pDetails)
    output("        Number new resets:                 %llu", pDetails->numResets)
    output("        Number new programming errors:     %llu", pDetails->
↪numProgrammingErrors)
    output("        Number new driver errors:          %llu", pDetails->
↪numDriverErrors)
```

(continues on next page)

(continued from previous page)

```
    output("        Number new compute errors:        %llu", pDetails->
↪numComputeErrors)
    output("        Number new non-compute errors:    %llu", pDetails->
↪numNonComputeErrors)
    output("        Number new cache errors:          %llu", pDetails->numCacheErrors)
    output("        Number new memory errors:         %llu", pDetails->
↪numMemoryErrors)
    output("        Number new PCI errors:            %llu", pDetails->numPciErrors)
    output("        Number new fabric errors:         %llu", pDetails->
↪numFabricErrors)
    output("        Number new display errors:        %llu", pDetails->
↪numDisplayErrors)
```

10.14 Performing diagnostics

Diagnostics is the process of taking a device offline and requesting that the hardware run self-checks and repairs. This is achieved using the function ::zetSysmanDiagnosticsRunTests(). On return from the function, software can use the diagnostics return code (::zet_diag_result_t) to determine the new course of action:

1. ::ZET_DIAG_RESULT_NO_ERRORS - No errors found and workloads can resume submission to the hardware.
2. ::ZET_DIAG_RESULT_ABORT - Hardware had problems running diagnostic tests.
3. ::ZET_DIAG_RESULT_FAIL_CANT_REPAIR - Hardware had problems setting up repair. Card should be removed from the system.
4. ::ZET_DIAG_RESULT_REBOOT_FOR_REPAIR - Hardware has prepared for repair and requires a reboot after which time workloads can resume submission.

The function ::zetSysmanDeviceGetRepairStatus() can be used to determine if the device has been repaired.

There are multiple diagnostic test suites that can be run and these are defined in the enumerator ::zet_diag_type_t. The function ::zetSysmanDiagnosticsGet() will enumerate each available test suite and the function ::zetSysmanDiagnosticsGetProperties() can be used to determine the type and name of each test suite (::zet_diag_properties_t.type and ::zet_diag_properties_t.type).

Each test suite contains one or more diagnostic tests. On some systems, it is possible to run only a subset of the tests. Use the function ::zetSysmanDiagnosticsGetProperties() and check that ::zet_diag_properties_t.haveTests is true to determine if this feature is available. If it is, the function ::zetSysmanDiagnosticsGetTests() can be called to get the list of individual tests that can be run.

When running diagnostics for a test suite using ::zetSysmanDiagnosticsRunTests(), it is possible to specify the start and index of tests in the suite. Setting to ::ZET_DIAG_FIRST_TEST_INDEX and ::ZET_DIAG_LAST_TEST_INDEX will run all tests in the suite. If it is possible to run a subset of tests, specify the index of the start test and the end test - all tests that have an index in this range will be run.

The table below summaries all the diagnostic management functions:

Function	Description
::zetSysmanDiagnosticsGet()	Get handles to the available diagnostic test suites that can be run.
::zetSysmanDiagnosticsGetProperties()	Get information about a test suite - type, name, location and if individual tests can be run.
::zetSysmanDiagnosticsGetTests()	Get list of individual diagnostic tests that can be run.
::zetSysmanDiagnosticsRunTests()	Run either all or individual diagnostic tests.

The pseudo code below shows how to discover all test suites and the tests in each:

```
function ListDiagnosticTests(zet_sysman_handle_t hSysmanDevice)
{
    uint32_t numTestSuites
    if ((zetSysmanDiagnosticsGet(hSysmanDevice, &numTestSuites, NULL) == ZE_RESULT_
↪SUCCESS))
        zet_sysman_diag_handle_t* phTestSuites =
            allocate_memory(numTestSuites * sizeof(zet_sysman_diag_handle_t))
        if (zetSysmanDiagnosticsGet(hSysmanDevice, &numTestSuites, phTestSuites) ==␣
↪ZE_RESULT_SUCCESS)
            for (suiteIndex = 0 .. numTestSuites-1)
                uint32_t numTests = 0
                zet_diag_test_t* pTests
                zet_diag_properties_t suiteProps
                if (zetSysmanDiagnosticsGetProperties(phTestSuites[suiteIndex], &
↪suiteProps) != ZE_RESULT_SUCCESS)
                    next_loop(suiteIndex)
                output("Diagnostic test suite %s:", suiteProps.name)
                if (!suiteProps.haveTests)
                    output("    There are no individual tests that can be selected.")
                    next_loop(suiteIndex)
                if (zetSysmanDiagnosticsGetTests(phTestSuites[suiteIndex], &numTests,␣
↪NULL) != ZE_RESULT_SUCCESS)
                    output("    Problem getting list of individual tests.")
                    next_loop(suiteIndex)
                pTests = allocate_memory(numTests * sizeof(zet_diag_test_t*))
                if (zetSysmanDiagnosticsGetTests(phTestSuites[suiteIndex], &numTests,␣
↪pTests) != ZE_RESULT_SUCCESS)
                    output("    Problem getting list of individual tests.")
                    next_loop(suiteIndex)
                for (i = 0 .. numTests-1)
                    output("    Test %u: %s", pTests[i].index, pTests[i].name)
    free_memory(...)
```

10.15 Events

Events are a way to determine if changes have occurred on a device e.g. new RAS errors. An application registers the events that it wishes to receive notification about and then it queries to receive notifications. The query can request a blocking wait - this will put the calling application thread to sleep until new notifications are received.

For every device on which the application wants to receive events, it should perform the following actions:

1. Use ::zetSysmanEventGet() to get an event handler from the Sysman handle for the device.
2. Use ::zetSysmanEventSetConfig() to indicate which events it wasnts to listen to.

3. For each event, call the appropriate function to set conditions that will trigger the event.

Finally, the application calls ::zetSysmanEventListen() with a list of event handles that it wishes to listen for events on. A wait timeout is used to request non-blocking operations (timeout = ::ZET_EVENT_WAIT_NONE) or blocking operations (timeout = ::ZET_EVENT_WAIT_INFINITE) or to return after a specified amount of time even if no events have been received.

Once events have occurred, the application can call ::zetSysmanEventGetState() to determine the list of events that have been received for each event handle. If events have been received, the application can use the function relevant to the event to determine the actual state.

The list of events is given in the table below. For each event, the corresponding configuration and state functions are shown. Where a configuration function is not shown, the event is generated automatically; where a configuration function is shown, it must be called to enable the event and/or provide threshold conditions.

Event	Trigger	Configuration function	State function
::ZET_SYSMAN_EV ENT_TYPE_DEVICE _RESET	Device is about to be reset by the driver		
::ZET_SYSMAN_EV ENT_TYPE_DEVICE _SLEEP_STATE_EN TER	Device is about to enter a deep sleep state		
::ZET_SYSMAN_EV ENT_TYPE_DEVICE _SLEEP_STATE_EX IT	Device is exiting a deep sleep state		
::ZET_SYSMAN_EV ENT_TYPE_FREQ_T HROTTLED	Frequency starts being throttled		::zetSysmanFreq uencyGetState()
::ZET_SYSMAN_EV ENT_TYPE_ENERGY _THRESHOLD_CROS SED	Energy consumption threshold is reached	::zetSysmanPowe rSetEnergyThres hold()	
::ZET_SYSMAN_EV ENT_TYPE_TEMP_C RITICAL	Critical temperature is reached	::zetSysmanTemp eratureSetConfi g()	::zetSysmanTemp eratureGetState ()
::ZET_SYSMAN_EV ENT_TYPE_TEMP_T HRESHOLD1	Temperature crosses threshold 1	::zetSysmanTemp eratureSetConfi g()	::zetSysmanTemp eratureGetState ()
::ZET_SYSMAN_EV ENT_TYPE_TEMP_T HRESHOLD2	Temperature crosses threshold 2	::zetSysmanTemp eratureSetConfi g()	::zetSysmanTemp eratureGetState ()
::ZET_SYSMAN_EV ENT_TYPE_MEM_HE ALTH	Health of device memory changes		::zetSysmanMemo ryGetState()
::ZET_SYSMAN_EV ENT_TYPE_FABRIC _PORT_HEALTH	Health of fabric ports change		::zetSysmanFabr icPortGetState( )
::ZET_SYSMAN_EV ENT_TYPE_RAS_CO RRECTABLE_ERROR S	RAS correctable errors cross thresholds	::zetSysmanRasS etConfig()	::zetSysmanRasG etState()
::ZET_SYSMAN_EV ENT_TYPE_RAS_UN CORRECTABLE_ERR ORS	RAS uncorrectable errors cross thresholds	::zetSysmanRasS etConfig()	::zetSysmanRasG etState()

The call to ::zetSysmanEventListen() requires the driver handle. The list of event handles must only be for devices that have been enumerated from that driver, otherwise and error will be returned. If the application is managing devices from multiple drivers, it will need to call this function separately for each driver.

The table below summaries all the event management functions:

Function	Description
::zetSysmanEventGet()	Get the event handle for a specific Sysman device.
::zetSysmanEventGetConfig()	Get the current list of events for a given event handle that have been registered.
::zetSysmanEventSetConfig()	Set the events that should be registered on a given event handle.
::zetSysmanEventGetState()	Get the list of events that have been received for a given event handle.
::zetSysmanEventListen()	Wait for events to arrive for a given list of event handles.

The pseudo code below shows how to configure all temperature sensors to trigger an event when the temperature exceeds a specified threshold or when the critical temperature is reached.

```
function WaitForExcessTemperatureEvent(zet_driver_handle_t hDriver, double tempLimit)
{
    # This will contain the number of event handles (devices) that we will listen for␣
↪events from
    var numEventHandles = 0

    # Get list of all devices under this driver
    uint32_t deviceCount = 0
    zeDeviceGet(hDriver, &deviceCount, nullptr)
    # Allocate memory for all device handles
    ze_device_handle_t* phDevices =
        allocate_memory(deviceCount * sizeof(ze_device_handle_t))
    # Allocate memory for the event handle for each device
    zet_sysman_event_handle_t* phEvents =
        allocate_memory(deviceCount * sizeof(zet_sysman_event_handle_t))
    # Allocate memory for the event handles that we will actually listen to
    zet_sysman_event_handle_t* phListenEvents =
        allocate_memory(deviceCount * sizeof(zet_sysman_event_handle_t))
    # Allocate memory so that we can map an event handle in phListenEvent to the␣
↪device handle
    uint32_t* pListenDeviceIndex = allocate_memory(deviceCount * sizeof(uint32_t))

    # Get all device handles
    zeDeviceGet(hDriver, &deviceCount, phDevices)
    for(devIndex = 0 .. deviceCount-1)
        # Get Sysman handle for the device
        zet_sysman_handle_t hSysmanDevice
        if (zetSysmanGet(phDevices[devIndex], ZET_SYSMAN_VERSION_CURRENT, &
↪hSysmanDevice)
                != ZE_RESULT_SUCCESS)
                    next_loop(devIndex)

        # Get event handle for this device
        if (zetSysmanEventGet(hSysmanDevice, &phEvents[devIndex]) != ZE_RESULT_
↪SUCCESS)
            next_loop(devIndex)

        # Get handles to all temperature sensors
        uint32_t numTempSensors = 0
        if (zetSysmanTemperatureGet(hSysmanDevice, &numTempSensors, NULL) != ZE_
↪RESULT_SUCCESS)
            next_loop(devIndex)
        zet_sysman_temp_handle_t* allTempSensors
            allocate_memory(deviceCount * sizeof(zet_sysman_temp_handle_t))
        if (zetSysmanTemperatureGet(hSysmanDevice, &numTempSensors, allTempSensors)␣
↪== ZE_RESULT_SUCCESS)
```

(continues on next page)

(continued from previous page)

```
            # Configure each temperature sensor to trigger a critical event and a␣
↪threshold1 event
            var numConfiguredTempSensors = 0
            for (tempIndex = 0 .. numTempSensors-1)
                if (zetSysmanTemperatureGetConfig(allTempSensors[tempIndex], &config)␣
↪!= ZE_RESULT_SUCCESS)
                    next_loop(tempIndex)
                zet_temp_config_t config
                config.enableCritical = true
                config.threshold1.enableHighToLow = false
                config.threshold1.enableLowToHigh = true
                config.threshold1.threshold = tempLimit
                config.threshold2.enableHighToLow = false
                config.threshold2.enableLowToHigh = false
                if (zetSysmanTemperatureSetConfig(allTempSensors[tempIndex], &config)␣
↪== ZE_RESULT_SUCCESS)
                    numConfiguredTempSensors++

        # If we configured any sensors to generate events, we can now register to␣
↪receive on this device
        if (numConfiguredTempSensors)
            zet_event_config_t eventConfig
            eventConfig.registered =
                ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL | ZET_SYSMAN_EVENT_TYPE_TEMP_
↪THRESHOLD1
            if (zetSysmanEventSetConfig(phEvents[devIndex], &eventConfig) == ZE_
↪RESULT_SUCCESS)
                phListenEvents[numEventHandles] = phEvents[devIndex]
                pListenDeviceIndex[numEventHandles] = devIndex
                numEventHandles++

    # If we registered to receive events on any devices, start listening now
    if (numEventHandles)
        # Block until we receive events
        uint32_t events
        if (zetSysmanEventListen(hDriver, ZET_EVENT_WAIT_INFINITE, deviceCount,␣
↪phListenEvents, &events)
            == ZE_RESULT_SUCCESS)
                for (evtIndex .. numEventHandles)
                    if (zetSysmanEventGetState(phListenEvents[evtIndex], true, &
↪events)
                        != ZE_RESULT_SUCCESS)
                            next_loop(evtIndex)
                    if (events & ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL)
                        output("Device %u: Went above the critical temperature.",
                            pListenDeviceIndex[evtIndex])
                    else if (events & ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1)
                        output("Device %u: Went above the temperature threshold %f.",
                            pListenDeviceIndex[evtIndex], tempLimit)

    free_memory(...)
```

CHAPTER

ELEVEN

SECURITY

11.1 Linux

The default security provided by the accelerator driver is to permit querying and controlling of system resources to the UNIX user **root**, querying only for users that are members of the UNIX group **root** and no access to any other user. Some queries are permitted from any user (e.g requesting current frequency, checking standby state).

It is the responsibility of the Linux distribution or the systems administrator to relax or tighten these permissions. This is typically done by adding udev daemon rules. For example, many distributions of Linux have the following rule:

```
root    video   /dev/dri/card0
```

This will permit all users in the UNIX group **video** to query information about system resources. In order to open up control access to users of the video group, udev rules need to be added for each relevant control. For example, to permit someone in the video group to disable standby, the following udev daemon rule would be needed:

```
chmod g+w /sys/class/drm/card0/rc6_enable
```

The full list of sysfs files used by the API are described in the table below. For each file, the list of affected API functions is given.

sysfs file	Description	Functions
/sys/class/drm/card0/rc6_enable	Used to enable/disable standby.	::zetSysmanStandbyGet() ::zetSysmanStandbyGetProperties() ::zetSysmanStandbyGetMode() ::zetSysmanStandbySetMode()
TBD	In development	TBD

11.2 Windows

At this time, Level0 Sysman does not support Windows.

11.3 Privileged telemetry

Certain telemetry makes a system vulnerable to side-channel attacks. By default, these will only be available to the administrator user on the system. It is up to the administrator to relax those requirements, as described in the preceding sections. This is the case for the following API calls:

Function	Description
::zetSysmanPciGetStats()	Access to total PCI throughput and number of packets can reveal useful information about the workload
::zetSysmanMemoryGetBandwidth()	Access to real-time device local memory bandwidth can reveal useful information about the workload
::zetSysmanFabricPortGetThroughput()	Access to real-time fabric data bandwidth can reveal useful information about the workload

11.4 Privileged controls

Certain controls can be used in denial-of-service attacks. By default, these will only be available to the administrator user on the system. It is up to the administrator to relax those requirements, as described in the preceding sections. This is the case for the following API calls:

Function	Description
::zetSysmanDeviceReset()	Device resets cause loss of data for running workloads.
::zetSysmanFirmwareGet()	All firmware operations must be handled with care.
::zetSysmanFirmwareGetProperties()	All firmware operations must be handled with care.
::zetSysmanFirmwareGetChecksum()	All firmware operations must be handled with care.
::zetSysmanFirmwareFlash()	All firmware operations must be handled with care.
::zetSysmanFabricPortSetConfig()	Putting fabric ports offline can distrupt workloads, causing uncorrectable errors.
::zetSysmanDiagnosticsRunTests()	Diagnostics take a device offline.

11.5 Virtualization

In virtualization environments, only the host is permitted to access any features of the API. Attempts to use the API in virtual machines will fail.

11.6 Function summary

The table below summarizes the default permissions for each API function:

Function	Administrator access	Group access	Other access	Virtual machi
::zetSysmanDeviceGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanDeviceWasRepaired()	read-only	read-only	read-only	no-access
::zetSysmanSchedulerGetCurrentMode()	read-only	read-only	read-only	no-access

continues on next pag

Table 1 – continued from previous page

Function	Administrator access	Group access	Other access	Virtual machi
::zetSysmanSchedulerGetTimeoutModeProperties()	read-only	read-only	read-only	no-access
::zetSysmanSchedulerGetTimesliceModeProperties()	read-only	read-only	read-only	no-access
::zetSysmanSchedulerSetTimeoutMode()	read-write	read-write	read-only	no-access
::zetSysmanSchedulerSetTimesliceMode()	read-write	read-write	read-only	no-access
::zetSysmanSchedulerSetExclusiveMode()	read-write	read-write	read-only	no-access
::zetSysmanSchedulerSetComputeUnitDebugMode()	read-write	read-write	read-only	no-access
::zetSysmanDeviceReset()	read-write	no-access	no-access	no-access
::zetSysmanPciGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanPciGetState()	read-only	read-only	read-only	no-access
::zetSysmanPciGetBars()	read-only	read-only	read-only	no-access
::zetSysmanPciGetStats	read-only	no-access	no-access	no-access
::zetSysmanPowerGet()	read-only	read-only	read-only	no-access
::zetSysmanPowerGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanPowerGetEnergyCounter()	read-only	read-only	read-only	no-access
::zetSysmanPowerGetEnergyThreshold()	read-only	read-only	no-access	no-access
::zetSysmanPowerSetEnergyThreshold()	read-write	read-write	no-access	no-access
::zetSysmanPowerGetLimits()	read-only	read-only	read-only	no-access
::zetSysmanPowerSetLimits()	read-write	read-write	read-only	no-access
::zetSysmanFrequencyGet()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyGetAvailableClocks()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyGetRange()	read-only	read-only	read-only	no-access
::zetSysmanFrequencySetRange()	read-write	read-write	read-only	no-access
::zetSysmanFrequencyGetState()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyGetThrottleTime()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyOcGetCapabilities()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyOcGetConfig()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyOcSetConfig()	read-write	no-access	no-access	no-access
::zetSysmanFrequencyOcGetIccMax()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyOcSetIccMax()	read-write	no-access	no-access	no-access
::zetSysmanFrequencyOcGetTjMax()	read-only	read-only	read-only	no-access
::zetSysmanFrequencyOcSetTjMax()	read-write	no-access	no-access	no-access
::zetSysmanEngineGet()	read-only	read-only	read-only	no-access
::zetSysmanEngineGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanEngineGetActivity()	read-only	read-only	read-only	no-access
::zetSysmanStandbyGet()	read-only	read-only	read-only	no-access
::zetSysmanStandbyGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanStandbyGetMode()	read-only	read-only	read-only	no-access
::zetSysmanStandbySetMode()	read-write	read-write	read-only	no-access
::zetSysmanFirmwareGet()	read-only	no-access	no-access	no-access
::zetSysmanFirmwareGetProperties()	read-only	no-access	no-access	no-access
::zetSysmanFirmwareGetChecksum()	read-only	no-access	no-access	no-access
::zetSysmanFirmwareFlash()	read-write	no-access	no-access	no-access
::zetSysmanMemoryGet()	read-only	read-only	read-only	no-access
::zetSysmanMemoryGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanMemoryGetBandwidth()	read-only	no-access	no-access	no-access
::zetSysmanMemoryGetState()	read-only	read-only	read-only	no-access
::zetSysmanFabricPortGet()	read-only	read-only	read-only	no-access
::zetSysmanFabricPortGetProperties()	read-only	read-only	read-only	no-access

continues on next pa

Table 1 – continued from previous page

Function	Administrator access	Group access	Other access	Virtual machi
::zetSysmanFabricPortGetLinkType()	read-only	read-only	read-only	no-access
::zetSysmanFabricPortGetConfig()	read-only	read-only	read-only	no-access
::zetSysmanFabricPortSetConfig()	read-write	no-access	no-access	no-access
::zetSysmanFabricPortGetState()	read-only	read-only	read-only	no-access
::zetSysmanFabricPortGetThroughput()	read-only	no-access	no-access	no-access
::zetSysmanTemperatureGet()	read-only	read-only	read-only	no-access
::zetSysmanTemperatureGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanTemperatureGetConfig()	read-only	read-only	no-access	no-access
::zetSysmanTemperatureSetConfig()	read-write	read-write	no-access	no-access
::zetSysmanTemperatureGetState()	read-only	read-only	read-only	no-access
::zetSysmanPsuGet()	read-only	read-only	read-only	no-access
::zetSysmanPsuGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanPsuGetState()	read-only	read-only	read-only	no-access
::zetSysmanFanGet()	read-only	read-only	read-only	no-access
::zetSysmanFanGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanFanGetConfig()	read-only	read-only	read-only	no-access
::zetSysmanFanSetConfig()	read-write	read-write	read-only	no-access
::zetSysmanFanGetState()	read-only	read-only	read-only	no-access
::zetSysmanLedGet()	read-only	read-only	read-only	no-access
::zetSysmanLedGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanLedGetState()	read-only	read-only	read-only	no-access
::zetSysmanLedSetState()	read-write	read-write	read-only	no-access
::zetSysmanRasGet()	read-only	read-only	read-only	no-access
::zetSysmanRasGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanRasGetConfig()	read-only	read-only	read-only	no-access
::zetSysmanRasSetConfig()	read-write	read-write	no-access	no-access
::zetSysmanRasGetState()	read-write	read-write	read-only	no-access
::zetSysmanEventGet()	read-only	read-only	read-only	no-access
::zetSysmanEventGetConfig()	read-only	read-only	read-only	no-access
::zetSysmanEventSetConfig()	read-write	read-write	read-write	no-access
::zetSysmanEventGetState()	read-only	read-only	read-only	no-access
::zetSysmanEventListen()	read-only	read-only	read-only	no-access
::zetSysmanDiagnosticsGet()	read-only	read-only	read-only	no-access
::zetSysmanDiagnosticsGetProperties()	read-only	read-only	read-only	no-access
::zetSysmanDiagnosticsGetTests()	read-only	read-only	read-only	no-access
::zetSysmanDiagnosticsRunTests()	read-write	no-access	no-access	no-access

CHAPTER

TWELVE

INTRODUCTION

SPIR-V is an open, royalty-free, standard intermediate language capable of representing parallel compute kernels. SPIR-V is adaptable to multiple execution environments: a SPIR-V module is consumed by an execution environment, as specified by a client API. This document describes the SPIR-V execution environment for the ‘One API’ Level-Zero API. The SPIR-V execution environment describes required support for some SPIR-V capabilities, additional semantics for some SPIR-V instructions, and additional validation rules that a SPIR-V binary module must adhere to in order to be considered valid.

This document is written for compiler developers who are generating SPIR-V modules intended to be consumed by the ‘One API’ Level-Zero API, for implementors of the ‘One API’ Level-Zero API, and for software developers who are using SPIR-V modules with the ‘One API’ Level-Zero API.

CHAPTER

THIRTEEN

COMMON PROPERTIES

This section describes common properties of all 'One API' Level-Zero environments that consume SPIR-V modules.

A SPIR-V module is interpreted as a series of 32-bit words in host endianness, with literal strings packed as described in the SPIR-V specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in the SPIR-V specification.

13.1 Supported SPIR-V Versions

The maximum SPIR-V version supported by a device is described by ::ze_device_kernel_properties_t.spirvVersionSupported.

13.2 Extended Instruction Sets

The **OpenCL.std** extended instruction set for OpenCL is supported.

13.3 Source Language Encoding

The source language version is purely informational and has no semantic meaning.

13.4 Numerical Type Formats

Floating-point types are represented and stored using IEEE-754 semantics. All integer formats are represented and stored using 2's-complement format.

13.5 Supported Types

The following types are supported. Note that some types may require additional capabilities, and may not be supported by all environments.

13.5.1 Basic Scalar and Vector Types

OpTypeVoid is supported.

The following scalar types are supported:

- **OpTypeBool**
- **OpTypeInt**, with *Width* equal to 8, 16, 32, or 64, and with *Signedness* equal to zero, indicating no signedness semantics.
- **OpTypeFloat**, with *Width* equal to 16, 32, or 64.

OpTypeVector vector types are supported. The vector *Component Type* may be any of the scalar types described above. Supported vector *Component Counts* are 2, 3, 4, 8, or 16.

OpTypeArray array types are supported, **OpTypeStruct** struct types are supported, **OpTypeFunction** functions are supported, and **OpTypePointer** pointer types are supported.

13.5.2 Image-Related Data Types

The following table describes the supported **OpTypeImage** image types:

Dim	*Depth*	*Arrayed*	**Description**
1D	`0`	`0`	A 1D image.
1D	`0`	`1`	A 1D image array.
2D	`0`	`0`	A 2D image.
2D	`1`	`0`	A 2D depth image.
2D	`0`	`1`	A 2D image array.
2D	`1`	`1`	A 2D depth image array.
3D	`0`	`0`	A 3D image.
Buffer	`0`	`0`	A 1D buffer image.

OpTypeSampler sampler typed are supported.

13.6 Kernels

An **OpFunction** in a SPIR-V module that is identified with **OpEntryPoint** defines a kernel that may be launched using host API interfaces.

13.7 Kernel Return Types

The *Result Type* for an **OpFunction** identified with **OpEntryPoint** must be **OpTypeVoid**.

13.8 Kernel Arguments

An **OpFunctionParameter** for an **OpFunction** that is identified with **OpEntryPoint** defines a kernel argument. Allowed types for kernel arguments are:

- **OpTypeInt**
- **OpTypeFloat**
- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**
- **OpTypeSampler**
- **OpTypeImage**

For **OpTypeInt** parameters, supported *Widths* are 8, 16, 32, and 64, and must have no signedness semantics.

For **OpTypeFloat** parameters, supported *Widths* are 16 and 32.

For **OpTypeStruct** parameters, supported structure *Member Types* are:

- **OpTypeInt**
- **OpTypeFloat**
- **OpTypeStruct**
- **OpTypeVector**
- **OpTypePointer**

For **OpTypePointer** parameters, supported *Storage Classes* are:

- **CrossWorkgroup**
- **Workgroup**
- **UniformConstant**

Environments that support extensions or optional features may allow additional types in an entry point's parameter list.

CHAPTER FOURTEEN

REQUIRED CAPABILITIES

14.1 SPIR-V 1.0

An environment that supports SPIR-V 1.0 must support SPIR-V 1.0 modules that declare the following capabilities:

- **Addresses**
- **Float16Buffer**
- **Int64**
- **Int16**
- **Int8**
- **Kernel**
- **Linkage**
- **Vector16**
- **GenericPointer**
- **Groups**
- **ImageBasic** (for devices supporting ::ze_device_image_properties_t.supported)
- **Float16** (for devices supporting ::ze_device_kernel_properties_t.fp16Supported)
- **Float64** (for devices supporting ::ze_device_kernel_properties_t.fp64Supported)
- **Int64Atomics** (for devices supporting ::ze_device_kernel_properties_t.int64AtomicsSupported)

If the 'One API' environment supports the **ImageBasic** capability, then the following capabilities must also be supported:

- **LiteralSampler**
- **Sampled1D**
- **Image1D**
- **SampledBuffer**
- **ImageBuffer**
- **ImageReadWrite**

14.2 SPIR-V 1.1

An environment supporting SPIR-V 1.1 must support SPIR-V 1.1 modules that declare the capabilities required for SPIR-V 1.0 modules, above.

SPIR-V 1.1 does not add any new required capabilities.

14.3 SPIR-V 1.2

An environment supporting SPIR-V 1.2 must support SPIR-V 1.2 modules that declare the capabilities required for SPIR-V 1.1 modules, above.

SPIR-V 1.2 does not add any new required capabilities.

CHAPTER FIFTEEN

VALIDATION RULES

The following are a list of validation rules that apply to SPIR-V modules executing in all ‘One API’ Level-Zero environments:

The *Execution Model* declared in **OpEntryPoint** must be **Kernel**.

The *Addressing Model* declared in **OpMemoryModel** must **Physical64**, indicating that device pointers are 64-bits.

The *Memory Model* declared in **OpMemoryModel** must be **OpenCL**.

For all **OpTypeInt** integer type-declaration instructions:

- *Signedness* must be 0, indicating no signedness semantics.

For all **OpTypeImage** type-declaration instructions: * *Sampled Type* must be **OpTypeVoid**. * *Sampled* must be 0, indicating that the image usage will be known at run time, not at compile time. * *MS* must be 0, indicating single-sampled content. * *Arrayed* may only be set to 1, indicating arrayed content, when *Dim* is set to **1D** or **2D**. * *Image Format* must be **Unknown**, indicating that the image does not have a specified format. * The optional image *Access Qualifier* must be present.

The image write instruction **OpImageWrite** must not include any optional *Image Operands*.

The image read instructions **OpImageRead** and **OpImageSampleExplicitLod** must not include the optional *Image Operand* **ConstOffset**.

For all *Atomic Instructions*:

- 32-bit integer types are supported for the *Result Type* and/or type of *Value*. 64-bit integer types are optionally supported for the *Result Type* and/or type of *Value* for devices supporting ::ze_device_kernel_properties_t.int64AtomicsSupported.
- The *Pointer* operand must be a pointer to the **Function**, **Workgroup**, **CrossWorkGroup**, or **Generic** *Storage Classes*.

Recursion is not supported. The static function call graph for an entry point must not contain cycles.

Whether irreducible control flow is legal is implementation defined.

For the instructions **OpGroupAsyncCopy** and **OpGroupWaitEvents**, *Scope* for *Execution* must be:

- **Workgroup**

For all other instructions, *Scope* for *Execution* must be one of:

- **Workgroup**
- **Subgroup**

Scope for *Memory* must be one of:

- **CrossDevice**

- **Device**
- **Workgroup**
- **Invocation**
- **Subgroup**

CHAPTER

SIXTEEN

EXTENSIONS

16.1 `SPV_INTEL_subgroups`

'One API' Level-Zero API environments must accept SPIR-V modules that declare use of the `SPV_INTEL_subgroups` extension via **OpExtension**.

When use of the `SPV_INTEL_subgroups` extension is declared in the module via **OpExtension**, the environment must accept modules that declare the following SPIR-V capabilities:

- **SubgroupShuffleINTEL**
- **SubgroupBufferBlockIOINTEL**
- **SubgroupImageBlockIOINTEL**

The environment must accept the following types for *Data* for the **SubgroupShuffleINTEL** instructions:

- Scalars and **OpTypeVectors** with 2, 4, 8, or 16 *Component Count* components of the following *Component Type* types:
  - **OpTypeFloat** with a *Width* of 32 bits (`float`)
  - TBD: char types?
  - **OpTypeInt** with a *Width* of 16 bits and *Signedness* of 0 (`short` and `ushort`)
  - **OpTypeInt** with a *Width* of 32 bits and *Signedness* of 0 (`int` and `uint`)
- Scalars of **OpTypeInt** with a *Width* of 64 bits and *Signedness* of 0 (`long` and `ulong`)
  - TBD: vectors of long types?

Additionally, if the **Float16** capability is declared and supported:

- Scalars of **OpTypeFloat** with a *Width* of 16 bits (`half`)

Additionally, if the **Float64** capability is declared and supported:

- Scalars of **OpTypeFloat** with a *Width* of 64 bits (`double`)

The environment must accept the following types for *Result* and *Data* for the **SubgroupBufferBlockIOINTEL** and **SubgroupImageBlockIOINTEL** instructions:

- Scalars and **OpTypeVectors** with 2, 4, or 8 *Component Count* components of the following *Component Type* types:
  - **OpTypeInt** with a *Width* of 32 bits and *Signedness* of 0 (`int` and `uint`)
  - **OpTypeInt** with a *Width* of 16 bits and *Signedness* of 0 (`short` and `ushort`)

For *Ptr*, valid *Storage Classes* are:

- **CrossWorkGroup** (`global`)

For *Image*:

- *Dim* must be *2D*
- *Depth* must be 0 (not a depth image)
- *Arrayed* must be 0 (non-arrayed content)
- *MS* must be 0 (single-sampled content)

For *Coordinate*, the following types are supported:

- **OpTypeVectors** with two *Component Count* components of *Component Type* **OpTypeInt** with a *Width* of 32 bits and *Signedness* of 0 (`int2`)

16.1.1 Notes and Restrictions

The **SubgroupShuffleINTEL** instructions may be placed within non-uniform control flow and hence do not have to be encountered by all invocations in the subgroup, however *Data* may only be shuffled among invocations encountering the **SubgroupShuffleINTEL** instruction. Shuffling *Data* from an invocation that does not encounter the **SubgroupShuffleINTEL** instruction will produce undefined results.

There is no defined behavior for out-of-range shuffle indices for the **SubgroupShuffleINTEL** instructions.

The **SubgroupBufferBlockIOINTEL** and **SubgroupImageBlockIOINTEL** instructions are only guaranteed to work correctly if placed strictly within uniform control flow within the subgroup. This ensures that if any invocation executes it, all invocations will execute it. If placed elsewhere, behavior is undefined.

There is no defined out-of-range behavior for the **SubgroupBufferBlockIOINTEL** instructions.

The **SubgroupImageBlockIOINTEL** instructions do support bounds checking, however they bounds-check to the image width in units of `uints`, not in units of image elements. This means:

- If the image has an *Image Format* size equal to the size of a `uint` (four bytes, for example **Rgba8**), the image will be correctly bounds-checked. In this case, out-of-bounds reads will return the edge image element (the equivalent of **ClampToEdge**), and out-of-bounds writes will be ignored.
- If the image has an *Image Format* size less than the size of a `uint` (such as **R8**), the entire image is addressable, however bounds checking will occur too late. For this reason, extra care should be taken to avoid out-of-bounds reads and writes, since out-of-bounds reads may return invalid data and out-of-bounds writes may corrupt other images or buffers unpredictably.

The following restrictions apply to the **SubgroupBufferBlockIOINTEL** instructions:

- The pointer *Ptr* must be 32-bit (4-byte) aligned for reads, and must be 128-bit (16-byte) aligned for writes.

The following restrictions apply to the **SubgroupImageBlockIOINTEL** instructions:

- The behavior of the **SubgroupImageBlockIOINTEL** instructions is undefined for images with an element size greater than four bytes (such as **Rgba32f**).

The following restrictions apply to the **OpSubgroupImageBlockWriteINTEL** instruction:

- Unlike the image block read instruction, which may read from any arbitrary byte offset, the x-component of the byte coordinate for the image block write instruction must be a multiple of four; in other words, the write must begin at a 32-bit boundary. There is no restriction on the y-component of the coordinate.

16.2 Other Extensions to Consider:

- SPV_INTEL_media_block_io / cl_intel_spirv_media_block_io?

CHAPTER

SEVENTEEN

NUMERICAL COMPLIANCE

The ‘One API’ Level-Zero environment will meet or exceed the numerical compliance requirements defined in the OpenCL SPIR-V Environment Specification. See: Numerical Compliance.

CHAPTER

EIGHTEEN

IMAGE ADDRESSING AND FILTERING

The ‘One API’ Level-Zero environment image addressing and filtering behavior is compatible with the behavior defined in the OpenCL SPIR-V Environment Specification. See: Image Addressing and Filtering.

CHAPTER

NINETEEN

CORE API

oneAPI Level Zero Specification - Version 0.91

19.1 Common

- Enumerations
  - *ze_result_t*
- Structures
  - *ze_ipc_mem_handle_t*
  - *ze_ipc_event_pool_handle_t*

19.1.1 Common Enums

ze_result_t

enum ze_result_t
Defines Return/Error codes.

Values:

enumerator ZE_RESULT_SUCCESS = 0
[Core] success

enumerator ZE_RESULT_NOT_READY = 1
[Core] synchronization primitive not signaled

enumerator ZE_RESULT_ERROR_DEVICE_LOST = 0x70000001
[Core] device hung, reset, was removed, or driver update occurred

enumerator ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY
[Core] insufficient host memory to satisfy call

enumerator ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY
[Core] insufficient device memory to satisfy call

enumerator ZE_RESULT_ERROR_MODULE_BUILD_FAILURE
[Core] error occurred when building module, see build log for details

enumerator ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS = 0x70010000
[Tools] access denied due to permission level

enumerator ZE_RESULT_ERROR_NOT_AVAILABLE
[Tools] resource already in use and simultaneous access not allowed

enumerator ZE_RESULT_ERROR_UNINITIALIZED = 0x78000001
[Validation] driver is not initialized

enumerator ZE_RESULT_ERROR_UNSUPPORTED_VERSION
[Validation] generic error code for unsupported versions

enumerator ZE_RESULT_ERROR_UNSUPPORTED_FEATURE
[Validation] generic error code for unsupported features

enumerator ZE_RESULT_ERROR_INVALID_ARGUMENT
[Validation] generic error code for invalid arguments

enumerator ZE_RESULT_ERROR_INVALID_NULL_HANDLE
[Validation] handle argument is not valid

enumerator ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE
[Validation] object pointed to by handle still in-use by device

enumerator ZE_RESULT_ERROR_INVALID_NULL_POINTER
[Validation] pointer argument may not be nullptr

enumerator ZE_RESULT_ERROR_INVALID_SIZE
[Validation] size argument is invalid (e.g., must not be zero)

enumerator ZE_RESULT_ERROR_UNSUPPORTED_SIZE
[Validation] size argument is not supported by the device (e.g., too large)

enumerator ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT
[Validation] alignment argument is not supported by the device (e.g., too small)

enumerator ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT
[Validation] synchronization object in invalid state

enumerator ZE_RESULT_ERROR_INVALID_ENUMERATION
[Validation] enumerator argument is not valid

enumerator ZE_RESULT_ERROR_UNSUPPORTED_ENUMERATION
[Validation] enumerator argument is not supported by the device

enumerator ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT
[Validation] image format is not supported by the device

enumerator ZE_RESULT_ERROR_INVALID_NATIVE_BINARY
[Validation] native binary is not supported by the device

enumerator ZE_RESULT_ERROR_INVALID_GLOBAL_NAME
[Validation] global variable is not found in the module

enumerator ZE_RESULT_ERROR_INVALID_KERNEL_NAME
[Validation] kernel name is not found in the module

enumerator ZE_RESULT_ERROR_INVALID_FUNCTION_NAME
[Validation] function name is not found in the module

enumerator ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION
[Validation] group size dimension is not valid for the kernel or device

enumerator ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION
[Validation] global width dimension is not valid for the kernel or device

enumerator ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX
[Validation] kernel argument index is not valid for kernel

enumerator ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE
[Validation] kernel argument size does not match kernel

enumerator ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE
[Validation] value of kernel attribute is not valid for the kernel or device

enumerator ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE
[Validation] command list type does not match command queue type

enumerator ZE_RESULT_ERROR_OVERLAPPING_REGIONS
[Validation] copy operations do not support overlapping regions of memory

enumerator ZE_RESULT_ERROR_UNKNOWN = 0x7fffffff
[Core] unknown or internal error

19.1.2 Common Structures

ze_ipc_mem_handle_t

struct ze_ipc_mem_handle_t
IPC handle to a memory allocation.

Public Members

char **data**[**ZE_MAX_IPC_HANDLE_SIZE**]
Opaque data representing an IPC handle.

ze_ipc_event_pool_handle_t

struct ze_ipc_event_pool_handle_t
IPC handle to a event pool allocation.

Public Members

char **data**[**ZE_MAX_IPC_HANDLE_SIZE**]
Opaque data representing an IPC handle.

19.2 Driver

- Functions
  - *zeInit*
  - *zeDriverGet*
  - *zeDriverGetApiVersion*
  - *zeDriverGetProperties*
  - *zeDriverGetIPCProperties*
  - *zeDriverGetExtensionFunctionAddress*

- Enumerations
  - *ze_init_flag_t*
  - *ze_api_version_t*
  - *ze_driver_properties_version_t*
  - *ze_driver_ipc_properties_version_t*
- Structures
  - *ze_driver_uuid_t*
  - *ze_driver_properties_t*
  - *ze_driver_ipc_properties_t*

19.2.1 Driver Functions

zeInit

__ze_api_export *ze_result_t* __zecall **`zeInit`** (*ze_init_flag_t flags*)

Initialize the 'One API' driver and must be called before any other API function.

Parameters

- `flags`: initialization flags

- If this function is not called then all other functions will return *ZE_RESULT_ERROR_UNINITIALIZED*.
- Only one instance of a driver per process will be initialized.
- This function is thread-safe for scenarios where multiple libraries may initialize the driver simultaneously.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - flags
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*

zeDriverGet

__ze_api_export *ze_result_t* __zecall **`zeDriverGet`** (uint32_t **pCount*, ze_driver_handle_t **phDrivers*)

Retrieves driver instances.

Parameters

- `pCount`: pointer to the number of driver instances. if count is zero, then the loader will update the value with the total number of drivers available. if count is non-zero, then the loader will only retrieve that number of drivers. if count is larger than the number of drivers available, then the loader will update the value with the correct number of drivers available.
- `phDrivers`: [optional][range(0, *pCount)] array of driver instance handles

- A driver represents a collection of physical devices.
- The application may pass nullptr for pDrivers when only querying the number of drivers.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetPlatformIDs

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zeDriverGetApiVersion

__ze_api_export *ze_result_t* __zecall **`zeDriverGetApiVersion`** (ze_driver_handle_t *hDriver*, *ze_api_version_t* **version*)

Returns the API version supported by the specified driver.

Parameters

- `hDriver`: handle of the driver instance
- `version`: api version

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == version`

zeDriverGetProperties

__ze_api_export *ze_result_t* __zecall **`zeDriverGetProperties`**(ze_driver_handle_t *hDriver*, *ze_driver_properties_t* **pDriverProperties*)

Retrieves properties of the driver.

Parameters

- `hDriver`: handle of the driver instance
- `pDriverProperties`: query result for driver properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clGetPlatformInfo**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pDriverProperties`

zeDriverGetIPCProperties

__ze_api_export *ze_result_t* __zecall **`zeDriverGetIPCProperties`**(ze_driver_handle_t *hDriver*, *ze_driver_ipc_properties_t* **pIPCProperties*)

Retrieves IPC attributes of the driver.

Parameters

- `hDriver`: handle of the driver instance
- `pIPCProperties`: query result for IPC properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pIPCProperties`

zeDriverGetExtensionFunctionAddress

__ze_api_export *ze_result_t* __zecall **zeDriverGetExtensionFunctionAddress** (ze_driver_handle_t *hDriver*, **const** char **pFuncName*, void ***pfunc*)

Retrieves an extension function for the specified driver.

Parameters

- `hDriver`: handle of the driver instance
- `pFuncName`: name of the extension function
- `pfunc`: pointer to extension function

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clGetExtensionFunctionAddressForPlatform**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pFuncName`
  - `nullptr == pfunc`

19.2.2 Driver Enums

ze_init_flag_t

enum ze_init_flag_t

Supported initialization flags.

Values:

enumerator ZE_INIT_FLAG_NONE = 0

default behavior

enumerator ZE_INIT_FLAG_GPU_ONLY = ZE_BIT(0)

only initialize GPU drivers

ze_api_version_t

enum ze_api_version_t

Supported API versions.

- API versions contain major and minor attributes, use ZE_MAJOR_VERSION and ZE_MINOR_VERSION

Values:

enumerator ZE_API_VERSION_1_0 = ZE_MAKE_VERSION(0, 91)

0.91

ze_driver_properties_version_t

enum ze_driver_properties_version_t

API version of *ze_driver_properties_t*.

Values:

enumerator ZE_DRIVER_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_driver_ipc_properties_version_t

enum ze_driver_ipc_properties_version_t

API version of *ze_driver_ipc_properties_t*.

Values:

enumerator ZE_DRIVER_IPC_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

19.2.3 Driver Structures

ze_driver_uuid_t

struct ze_driver_uuid_t

Driver universal unique id (UUID)

Public Members

uint8_t **id**[**ZE_MAX_DRIVER_UUID_SIZE**]

Opaque data representing a driver UUID.

ze_driver_properties_t

`struct ze_driver_properties_t`

Driver properties queried using *zeDriverGetProperties*.

Public Members

ze_driver_properties_version_t **`version`**

[in] *ZE_DRIVER_PROPERTIES_VERSION_CURRENT*

ze_driver_uuid_t **`uuid`**

[out] universal unique identifier.

uint32_t **`driverVersion`**

[out] driver version The driver version is a non-zero, monotonically increasing value where higher values always indicate a more recent version.

ze_driver_ipc_properties_t

`struct ze_driver_ipc_properties_t`

IPC properties queried using *zeDriverGetIPCProperties*.

Public Members

ze_driver_ipc_properties_version_t **`version`**

[in] *ZE_DRIVER_IPC_PROPERTIES_VERSION_CURRENT*

ze_bool_t **`memsSupported`**

[out] Supports passing memory allocations between processes. See ::*zeDriverGetMemIpcHandle*.

ze_bool_t **`eventsSupported`**

[out] Supports passing events between processes. See ::*zeEventPoolGetIpcHandle*.

19.3 Device

- Functions
  - *zeDeviceGet*
  - *zeDeviceGetSubDevices*
  - *zeDeviceGetProperties*
  - *zeDeviceGetComputeProperties*
  - *zeDeviceGetKernelProperties*
  - *zeDeviceGetMemoryProperties*
  - *zeDeviceGetMemoryAccessProperties*
  - *zeDeviceGetCacheProperties*
  - *zeDeviceGetImageProperties*
  - *zeDeviceGetP2PProperties*
  - *zeDeviceCanAccessPeer*

  - *zeDeviceSetLastLevelCacheConfig*

- Enumerations
  - *ze_device_properties_version_t*
  - *ze_device_type_t*
  - *ze_device_compute_properties_version_t*
  - *ze_device_kernel_properties_version_t*
  - *ze_fp_capabilities_t*
  - *ze_device_memory_properties_version_t*
  - *ze_device_memory_access_properties_version_t*
  - *ze_memory_access_capabilities_t*
  - *ze_device_cache_properties_version_t*
  - *ze_device_image_properties_version_t*
  - *ze_device_p2p_properties_version_t*
  - *ze_cache_config_t*
- Structures
  - *ze_device_uuid_t*
  - *ze_device_properties_t*
  - *ze_device_compute_properties_t*
  - *ze_native_kernel_uuid_t*
  - *ze_device_kernel_properties_t*
  - *ze_device_memory_properties_t*
  - *ze_device_memory_access_properties_t*
  - *ze_device_cache_properties_t*
  - *ze_device_image_properties_t*
  - *ze_device_p2p_properties_t*

19.3.1 Device Functions

zeDeviceGet

__ze_api_export *ze_result_t* __zecall **zeDeviceGet**(ze_driver_handle_t *hDriver*, uint32_t **pCount*, ze_device_handle_t **phDevices*)

Retrieves devices within a driver.

Parameters

- `hDriver`: handle of the driver instance
- `pCount`: pointer to the number of devices. if count is zero, then the driver will update the value with the total number of devices available. if count is non-zero, then driver will only retrieve that number of devices. if count is larger than the number of devices available, then the driver will update the value with the correct number of devices available.

- `phDevices`: [optional][range(0, *pCount)] array of handle of devices

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zeDeviceGetSubDevices

__ze_api_export *ze_result_t* __zecall **`zeDeviceGetSubDevices`**(ze_device_handle_t *hDevice*, uint32_t **pCount*, ze_device_handle_t **phSubdevices*)

Retrieves a sub-device from a device.

Parameters

- `hDevice`: handle of the device object
- `pCount`: pointer to the number of sub-devices. if count is zero, then the driver will update the value with the total number of sub-devices available. if count is non-zero, then driver will only retrieve that number of sub-devices. if count is larger than the number of sub-devices available, then the driver will update the value with the correct number of sub-devices available.
- `phSubdevices`: [optional][range(0, *pCount)] array of handle of sub-devices

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clCreateSubDevices

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zeDeviceGetProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetProperties** (ze_device_handle_t *hDevice*, *ze_device_properties_t* **pDeviceProperties*)

Retrieves properties of the device.

Parameters

- `hDevice`: handle of the device
- `pDeviceProperties`: query result for device properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetDeviceInfo

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pDeviceProperties`

zeDeviceGetComputeProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetComputeProperties** (ze_device_handle_t *hDevice*, *ze_device_compute_properties_t* **pComputeProperties*)

Retrieves compute properties of the device.

Parameters

- `hDevice`: handle of the device
- `pComputeProperties`: query result for compute properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetDeviceInfo

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pComputeProperties`

zeDeviceGetKernelProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetKernelProperties**(ze_device_handle_t *hDevice*, *ze_device_kernel_properties_t* **pKernelProperties*)

Retrieves kernel properties of the device.

Parameters

- `hDevice`: handle of the device
- `pKernelProperties`: query result for kernel properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pKernelProperties`

zeDeviceGetMemoryProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetMemoryProperties**(ze_device_handle_t *hDevice*, uint32_t **pCount*, *ze_device_memory_properties_t* **pMemProperties*)

Retrieves local memory properties of the device.

Parameters

- `hDevice`: handle of the device
- `pCount`: pointer to the number of memory properties. if count is zero, then the driver will update the value with the total number of memory properties available. if count is non-zero, then driver will only retrieve that number of memory properties. if count is larger than the number of memory properties available, then the driver will update the value with the correct number of memory properties available.
- `pMemProperties`: [optional][range(0, *pCount)] array of query results for memory properties

- Properties are reported for each physical memory type supported by the device.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetDeviceInfo

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zeDeviceGetMemoryAccessProperties

__ze_api_export *ze_result_t* __zecall **`zeDeviceGetMemoryAccessProperties`**(ze_device_handle_t *hDevice*, *ze_device_memory_access_properties_t* **pMemAccessProperties*)

Retrieves memory access properties of the device.

Parameters

- `hDevice`: handle of the device
- `pMemAccessProperties`: query result for memory access properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetDeviceInfo

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pMemAccessProperties`

zeDeviceGetCacheProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetCacheProperties**(ze_device_handle_t *hDevice*, *ze_device_cache_properties_t* **pCacheProperties*)

Retrieves cache properties of the device.

Parameters

- `hDevice`: handle of the device
- `pCacheProperties`: query result for cache properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clGetDeviceInfo

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCacheProperties`

zeDeviceGetImageProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetImageProperties**(ze_device_handle_t *hDevice*, *ze_device_image_properties_t* **pImageProperties*)

Retrieves image X_DEVICE_MEMORY_ACCESS_PROPERTIES_VERSION_CURRENT of the device.

Parameters

- `hDevice`: handle of the device
- `pImageProperties`: query result for image properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

    - `nullptr == hDevice`
  - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pImageProperties`

zeDeviceGetP2PProperties

__ze_api_export *ze_result_t* __zecall **zeDeviceGetP2PProperties**(ze_device_handle_t *hDevice*, ze_device_handle_t *hPeerDevice*, *ze_device_p2p_properties_t* **pP2PProperties*)

Retrieves Peer-to-Peer properties between one device and a peer devices.

Parameters

- `hDevice`: handle of the device performing the access
- `hPeerDevice`: handle of the peer device with the allocation
- `pP2PProperties`: Peer-to-Peer properties between source and peer device

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
  - `nullptr == hPeerDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pP2PProperties`

zeDeviceCanAccessPeer

__ze_api_export *ze_result_t* __zecall **zeDeviceCanAccessPeer**(ze_device_handle_t *hDevice*, ze_device_handle_t *hPeerDevice*, ze_bool_t **value*)

Queries if one device can directly access peer device allocations.

Parameters

- `hDevice`: handle of the device performing the access
- `hPeerDevice`: handle of the peer device with the allocation
- `value`: returned access capability

- Any device can access any other device within a node through a scale-up fabric.
- The following are conditions for CanAccessPeer query.

  - If both device and peer device are the same then return true.
  - If both sub-device and peer sub-device are the same then return true.
  - If both are sub-devices and share the same parent device then return true.
  - If both device and remote device are connected by a direct or indirect scale-up fabric or over PCIe (same root complex or shared PCIe switch) then true.
  - If both sub-device and remote parent device (and vice-versa) are connected by a direct or indirect scale-up fabric or over PCIe (same root complex or shared PCIe switch) then true.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
  - `nullptr == hPeerDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == value`

zeDeviceSetLastLevelCacheConfig

__ze_api_export *ze_result_t* __zecall **`zeDeviceSetLastLevelCacheConfig`**(ze_device_handle_t *hDevice*, *ze_cache_config_t* *CacheConfig*)

Sets the preferred Last Level cache configuration for a device.

Parameters

- `hDevice`: handle of the device
- `CacheConfig`: CacheConfig

- The application may **not** call this function from simultaneous threads with the same device handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - CacheConfig

- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*

19.3.2 Device Enums

ze_device_properties_version_t

enum ze_device_properties_version_t
API version of *ze_device_properties_t*.

Values:

enumerator ZE_DEVICE_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_device_type_t

enum ze_device_type_t
Supported device types.

Values:

enumerator ZE_DEVICE_TYPE_GPU = 1
Graphics Processing Unit.

enumerator ZE_DEVICE_TYPE_FPGA
Field Programmable Gate Array.

ze_device_compute_properties_version_t

enum ze_device_compute_properties_version_t
API version of *ze_device_compute_properties_t*.

Values:

enumerator ZE_DEVICE_COMPUTE_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_device_kernel_properties_version_t

enum ze_device_kernel_properties_version_t
API version of *ze_device_kernel_properties_t*.

Values:

enumerator ZE_DEVICE_KERNEL_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_fp_capabilities_t

enum ze_fp_capabilities_t
Floating Point capabilities.

- floating-point capabilities of the device.

Values:

enumerator ZE_FP_CAPS_NONE = 0
None.

enumerator ZE_FP_CAPS_DENORM = ZE_BIT(0)
Supports denorms.

enumerator ZE_FP_CAPS_INF_NAN = ZE_BIT(1)
Supports INF and quiet NaNs.

enumerator ZE_FP_CAPS_ROUND_TO_NEAREST = ZE_BIT(2)
Supports rounding to nearest even rounding mode.

enumerator ZE_FP_CAPS_ROUND_TO_ZERO = ZE_BIT(3)
Supports rounding to zero.

enumerator ZE_FP_CAPS_ROUND_TO_INF = ZE_BIT(4)
Supports rounding to both positive and negative INF.

enumerator ZE_FP_CAPS_FMA = ZE_BIT(5)
Supports IEEE754-2008 fused multiply-add.

enumerator ZE_FP_CAPS_ROUNDED_DIVIDE_SQRT = ZE_BIT(6)
Supports rounding as defined by IEEE754 for divide and sqrt operations.

enumerator ZE_FP_CAPS_SOFT_FLOAT = ZE_BIT(7)
Uses software implementation for basic floating-point operations.

ze_device_memory_properties_version_t

enum ze_device_memory_properties_version_t
API version of *ze_device_memory_properties_t*.

Values:

enumerator ZE_DEVICE_MEMORY_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_device_memory_access_properties_version_t

enum ze_device_memory_access_properties_version_t
API version of *ze_device_memory_access_properties_t*.

Values:

enumerator ZE_DEVICE_MEMORY_ACCESS_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_memory_access_capabilities_t

enum ze_memory_access_capabilities_t

Memory access capabilities.

- Supported access capabilities for different types of memory allocations

Values:

enumerator ZE_MEMORY_ACCESS_NONE = 0

Access not supported.

enumerator ZE_MEMORY_ACCESS = ZE_BIT(0)

Supports load/store access.

enumerator ZE_MEMORY_ATOMIC_ACCESS = ZE_BIT(1)

Supports atomic access.

enumerator ZE_MEMORY_CONCURRENT_ACCESS = ZE_BIT(2)

Supports concurrent access.

enumerator ZE_MEMORY_CONCURRENT_ATOMIC_ACCESS = ZE_BIT(3)

Supports concurrent atomic access.

ze_device_cache_properties_version_t

enum ze_device_cache_properties_version_t

API version of *ze_device_cache_properties_t*.

Values:

enumerator ZE_DEVICE_CACHE_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_device_image_properties_version_t

enum ze_device_image_properties_version_t

API version of *ze_device_image_properties_t*.

Values:

enumerator ZE_DEVICE_IMAGE_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_device_p2p_properties_version_t

enum ze_device_p2p_properties_version_t

API version of *ze_device_p2p_properties_t*.

Values:

enumerator ZE_DEVICE_P2P_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_cache_config_t

enum ze_cache_config_t
Supported Cache Config.

- Supported Cache Config (Default, Large SLM, Large Data Cache)

Values:

enumerator ZE_CACHE_CONFIG_DEFAULT = ZE_BIT(0)
Default Config.

enumerator ZE_CACHE_CONFIG_LARGE_SLM = ZE_BIT(1)
Large SLM size.

enumerator ZE_CACHE_CONFIG_LARGE_DATA = ZE_BIT(2)
Large General Data size.

19.3.3 Device Structures

ze_device_uuid_t

struct ze_device_uuid_t
Device universal unique id (UUID)

Public Members

uint8_t **id**[**ZE_MAX_DEVICE_UUID_SIZE**]
Opaque data representing a device UUID.

ze_device_properties_t

struct ze_device_properties_t
Device properties queried using *zeDeviceGetProperties*.

Public Members

ze_device_properties_version_t **version**
[in] *ZE_DEVICE_PROPERTIES_VERSION_CURRENT*

ze_device_type_t **type**
[out] generic device type

uint32_t **vendorId**
[out] vendor id from PCI configuration

uint32_t **deviceId**
[out] device id from PCI configuration

ze_device_uuid_t **uuid**
[out] universal unique identifier.

ze_bool_t **isSubdevice**
[out] If the device handle used for query represents a sub-device.

uint32_t **subdeviceId**
: [out] sub-device id. Only valid if isSubdevice is true.

uint32_t **coreClockRate**
: [out] Clock rate for device core.

ze_bool_t **unifiedMemorySupported**
: [out] Supports unified physical memory between Host and device.

ze_bool_t **eccMemorySupported**
: [out] Supports error correction memory access.

ze_bool_t **onDemandPageFaultsSupported**
: [out] Supports on-demand page-faulting.

uint32_t **maxCommandQueues**
: [out] Maximum number of logical command queues.

uint32_t **numAsyncComputeEngines**
: [out] Number of asynchronous compute engines

uint32_t **numAsyncCopyEngines**
: [out] Number of asynchronous copy engines

uint32_t **maxCommandQueuePriority**
: [out] Maximum priority for command queues. Higher value is higher priority.

uint32_t **numThreadsPerEU**
: [out] Number of threads per EU.

uint32_t **physicalEUSimdWidth**
: [out] The physical EU simd width.

uint32_t **numEUsPerSubslice**
: [out] Number of EUs per sub-slice.

uint32_t **numSubslicesPerSlice**
: [out] Number of sub-slices per slice.

uint32_t **numSlices**
: [out] Number of slices.

uint64_t **timerResolution**
: [out] Returns the resolution of device timer in nanoseconds used for profiling, timestamps, etc.

char **name**[**ZE_MAX_DEVICE_NAME**]
: [out] Device name

ze_device_compute_properties_t

struct ze_device_compute_properties_t
: Device compute properties queried using *zeDeviceGetComputeProperties*.

Public Members

ze_device_compute_properties_version_t **version**
[in] *ZE_DEVICE_COMPUTE_PROPERTIES_VERSION_CURRENT*

uint32_t **maxTotalGroupSize**
[out] Maximum items per compute group. (maxGroupSizeX * maxGroupSizeY

- maxGroupSizeZ) <= maxTotalGroupSize

uint32_t **maxGroupSizeX**
[out] Maximum items for X dimension in group

uint32_t **maxGroupSizeY**
[out] Maximum items for Y dimension in group

uint32_t **maxGroupSizeZ**
[out] Maximum items for Z dimension in group

uint32_t **maxGroupCountX**
[out] Maximum groups that can be launched for x dimension

uint32_t **maxGroupCountY**
[out] Maximum groups that can be launched for y dimension

uint32_t **maxGroupCountZ**
[out] Maximum groups that can be launched for z dimension

uint32_t **maxSharedLocalMemory**
[out] Maximum shared local memory per group.

uint32_t **numSubGroupSizes**
[out] Number of subgroup sizes supported. This indicates number of entries in subGroupSizes.

uint32_t **subGroupSizes**[**ZE_SUBGROUPSIZE_COUNT**]
[out] Size group sizes supported.

ze_native_kernel_uuid_t

struct ze_native_kernel_uuid_t
Native kernel universal unique id (UUID)

Public Members

uint8_t **id**[**ZE_MAX_NATIVE_KERNEL_UUID_SIZE**]
Opaque data representing a native kernel UUID.

ze_device_kernel_properties_t

struct ze_device_kernel_properties_t
Device properties queried using *zeDeviceGetKernelProperties*.

Public Members

ze_device_kernel_properties_version_t **`version`**
: [in] *ZE_DEVICE_KERNEL_PROPERTIES_VERSION_CURRENT*

uint32_t **`spirvVersionSupported`**
: [out] Maximum supported SPIR-V version. Returns zero if SPIR-V is not supported. Contains major and minor attributes, use ZE_MAJOR_VERSION and ZE_MINOR_VERSION.

ze_native_kernel_uuid_t **`nativeKernelSupported`**
: [out] Compatibility UUID of supported native kernel. UUID may or may not be the same across driver release, devices, or operating systems. Application is responsible for ensuring UUID matches before creating module using previously created native kernel.

ze_bool_t **`fp16Supported`**
: [out] Supports 16-bit floating-point operations

ze_bool_t **`fp64Supported`**
: [out] Supports 64-bit floating-point operations

ze_bool_t **`int64AtomicsSupported`**
: [out] Supports 64-bit atomic operations

ze_bool_t **`dp4aSupported`**
: [out] Supports four component dot product and accumulate operations

ze_fp_capabilities_t **`halfFpCapabilities`**
: [out] Capabilities for half-precision floating-point operations.

ze_fp_capabilities_t **`singleFpCapabilities`**
: [out] Capabilities for single-precision floating-point operations.

ze_fp_capabilities_t **`doubleFpCapabilities`**
: [out] Capabilities for double-precision floating-point operations.

uint32_t **`maxArgumentsSize`**
: [out] Maximum kernel argument size that is supported.

uint32_t **`printfBufferSize`**
: [out] Maximum size of internal buffer that holds output of printf calls from kernel.

ze_device_memory_properties_t

`struct ze_device_memory_properties_t`
: Device local memory properties queried using *zeDeviceGetMemoryProperties*.

Public Members

ze_device_memory_properties_version_t **`version`**
: [in] *ZE_DEVICE_MEMORY_PROPERTIES_VERSION_CURRENT*

uint32_t **`maxClockRate`**
: [out] Maximum clock rate for device memory.

uint32_t **`maxBusWidth`**
: [out] Maximum bus width between device and memory.

uint64_t **`totalSize`**
: [out] Total memory size in bytes.

ze_device_memory_access_properties_t

struct ze_device_memory_access_properties_t
Device memory access properties queried using *zeDeviceGetMemoryAccessProperties*.

Public Members

ze_device_memory_access_properties_version_t **version**
[in] *ZE_DEVICE_MEMORY_ACCESS_PROPERTIES_VERSION_CURRENT*

ze_memory_access_capabilities_t **hostAllocCapabilities**
[out] Bitfield describing host memory capabilities

ze_memory_access_capabilities_t **deviceAllocCapabilities**
[out] Bitfield describing device memory capabilities

ze_memory_access_capabilities_t **sharedSingleDeviceAllocCapabilities**
[out] Bitfield describing shared (single-device) memory capabilities

ze_memory_access_capabilities_t **sharedCrossDeviceAllocCapabilities**
[out] Bitfield describing shared (cross-device) memory capabilities

ze_memory_access_capabilities_t **sharedSystemAllocCapabilities**
[out] Bitfield describing shared (system) memory capabilities

ze_device_cache_properties_t

struct ze_device_cache_properties_t
Device cache properties queried using *zeDeviceGetCacheProperties*.

Public Members

ze_device_cache_properties_version_t **version**
[in] *ZE_DEVICE_CACHE_PROPERTIES_VERSION_CURRENT*

ze_bool_t **intermediateCacheControlSupported**
[out] Support User control on Intermediate Cache (i.e. Resize SLM section vs Generic Cache)

size_t **intermediateCacheSize**
[out] Per-cache Intermediate Cache (L1/L2) size, in bytes

uint32_t **intermediateCachelineSize**
[out] Cacheline size in bytes for intermediate cacheline (L1/L2).

ze_bool_t **lastLevelCacheSizeControlSupported**
[out] Support User control on Last Level Cache (i.e. Resize SLM section vs Generic Cache).

size_t **lastLevelCacheSize**
[out] Per-cache Last Level Cache (L3) size, in bytes

uint32_t **lastLevelCachelineSize**
[out] Cacheline size in bytes for last-level cacheline (L3).

ze_device_image_properties_t

struct ze_device_image_properties_t
Device image properties queried using *zeDeviceGetComputeProperties*.

Public Members

ze_device_image_properties_version_t **version**
[in] *ZE_DEVICE_IMAGE_PROPERTIES_VERSION_CURRENT*

ze_bool_t **supported**
[out] Supports reading and writing of images. See ::*zeImageGetProperties* for format-specific capabilities.

uint32_t **maxImageDims1D**
[out] Maximum image dimensions for 1D resources.

uint32_t **maxImageDims2D**
[out] Maximum image dimensions for 2D resources.

uint32_t **maxImageDims3D**
[out] Maximum image dimensions for 3D resources.

uint64_t **maxImageBufferSize**
[out] Maximum image buffer size in bytes.

uint32_t **maxImageArraySlices**
[out] Maximum image array slices

uint32_t **maxSamplers**
[out] Max samplers that can be used in kernel.

uint32_t **maxReadImageArgs**
[out] Returns the maximum number of simultaneous image objects that can be read from by a kernel.

uint32_t **maxWriteImageArgs**
[out] Returns the maximum number of simultaneous image objects that can be written to by a kernel.

ze_device_p2p_properties_t

struct ze_device_p2p_properties_t
Device properties queried using *zeDeviceGetP2PProperties*.

Public Members

ze_device_p2p_properties_version_t **version**
[in] *ZE_DEVICE_P2P_PROPERTIES_VERSION_CURRENT*

ze_bool_t **accessSupported**
[out] Supports access between peer devices.

ze_bool_t **atomicsSupported**
[out] Supports atomics between peer devices.

19.4 Cmdqueue

- Functions
  - *zeCommandQueueCreate*
  - *zeCommandQueueDestroy*
  - *zeCommandQueueExecuteCommandLists*
  - *zeCommandQueueSynchronize*
- Enumerations
  - *ze_command_queue_desc_version_t*
  - *ze_command_queue_flag_t*
  - *ze_command_queue_mode_t*
  - *ze_command_queue_priority_t*
- Structures
  - *ze_command_queue_desc_t*

19.4.1 Cmdqueue Functions

zeCommandQueueCreate

__ze_api_export *ze_result_t* __zecall **zeCommandQueueCreate**(ze_device_handle_t *hDevice*, **const** *ze_command_queue_desc_t* **desc*, ze_command_queue_handle_t **phCommandQueue*)

Creates a command queue on the device.

Parameters

- `hDevice`: handle of the device object
- `desc`: pointer to command queue descriptor
- `phCommandQueue`: pointer to handle of command queue object created

- The command queue can only be used on the device on which it was created.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clCreateCommandQueue**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phCommandQueue`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
  - desc->mode
  - desc->priority
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeCommandQueueDestroy

__ze_api_export *ze_result_t* __zecall **`zeCommandQueueDestroy`**(ze_command_queue_handle_t *hCommandQueue*)

Destroys a command queue.

Parameters

- `hCommandQueue`: [release] handle of command queue object to destroy

- The application is responsible for making sure the device is not currently referencing the command queue before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this command queue
- The application may **not** call this function from simultaneous threads with the same command queue handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clReleaseCommandQueue**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandQueue`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeCommandQueueExecuteCommandLists

__ze_api_export *ze_result_t* __zecall **zeCommandQueueExecuteCommandLists**(ze_command_queue_handle_t *hCommandQueue*, uint32_t *numCommandLists*, ze_command_list_handle_t **phCommandLists*, ze_fence_handle_t *hFence*)

Executes a command list in a command queue.

Parameters

- `hCommandQueue`: handle of the command queue
- `numCommandLists`: number of command lists to execute
- `phCommandLists`: [range(0, numCommandLists)] list of handles of the command lists to execute
- `hFence`: [optional] handle of the fence to signal on completion

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- vkQueueSubmit

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandQueue`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phCommandLists`
- *ZE_RESULT_ERROR_INVALID_SIZE*
  - `0 == numCommandLists`
- *ZE_RESULT_ERROR_INVALID_COMMAND_LIST_TYPE*
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandQueueSynchronize

__ze_api_export *ze_result_t* __zecall **zeCommandQueueSynchronize** (ze_command_queue_handle_t *hCommandQueue*, uint32_t *timeout*)

Synchronizes a command queue by waiting on the host.

Parameters

- `hCommandQueue`: handle of the command queue
- `timeout`: if non-zero, then indicates the maximum time to yield before returning *ZE_RESULT_SUCCESS* or *ZE_RESULT_NOT_READY*; if zero, then operates exactly like *zeFenceQueryStatus*; if UINT32_MAX, then function will not return until complete or device is lost.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandQueue`
- *ZE_RESULT_NOT_READY*
  - timeout expired

19.4.2 Cmdqueue Enums

ze_command_queue_desc_version_t

enum ze_command_queue_desc_version_t

API version of *ze_command_queue_desc_t*.

Values:

enumerator ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_command_queue_flag_t

enum ze_command_queue_flag_t

Supported command queue flags.

Values:

enumerator ZE_COMMAND_QUEUE_FLAG_NONE = 0

default behavior

enumerator ZE_COMMAND_QUEUE_FLAG_COPY_ONLY = ZE_BIT(0)

command queue only supports enqueuing copy-only command lists

enumerator ZE_COMMAND_QUEUE_FLAG_LOGICAL_ONLY = ZE_BIT(1)
command queue is not tied to a physical command queue; driver may dynamically assign based on usage

enumerator ZE_COMMAND_QUEUE_FLAG_SINGLE_SLICE_ONLY = ZE_BIT(2)
'slice' size is device-specific. cannot be combined with COPY_ONLY.

command queue reserves and cannot consume more than a single slice.

enumerator ZE_COMMAND_QUEUE_FLAG_SUPPORTS_COOPERATIVE_KERNELS = ZE_BIT(3)
command queue supports command list with cooperative kernels. See *zeCommandListAppendLaunchCooperativeKernel* for more details. cannot be combined with COPY_ONLY.

ze_command_queue_mode_t

enum ze_command_queue_mode_t
Supported command queue modes.

Values:

enumerator ZE_COMMAND_QUEUE_MODE_DEFAULT = 0
implicit default behavior; uses driver-based heuristics

enumerator ZE_COMMAND_QUEUE_MODE_SYNCHRONOUS
Device execution always completes immediately on execute; Host thread is blocked using wait on implicit synchronization object

enumerator ZE_COMMAND_QUEUE_MODE_ASYNCHRONOUS
Device execution is scheduled and will complete in future; explicit synchronization object must be used to determine completeness

ze_command_queue_priority_t

enum ze_command_queue_priority_t
Supported command queue priorities.

Values:

enumerator ZE_COMMAND_QUEUE_PRIORITY_NORMAL = 0
[default] normal priority

enumerator ZE_COMMAND_QUEUE_PRIORITY_LOW
lower priority than normal

enumerator ZE_COMMAND_QUEUE_PRIORITY_HIGH
higher priority than normal

19.4.3 Cmdqueue Structures

ze_command_queue_desc_t

struct ze_command_queue_desc_t
Command Queue descriptor.

Public Members

ze_command_queue_desc_version_t **version**
[in] *ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT*

ze_command_queue_flag_t **flags**
[in] creation flags

ze_command_queue_mode_t **mode**
[in] operation mode

ze_command_queue_priority_t **priority**
[in] priority

uint32_t **ordinal**
[in] if logical-only flag is set, then will be ignored; if supports-cooperative-kernels is set, then may be ignored; else-if copy-only flag is set, then must be less than *ze_device_properties_t.numAsyncCopyEngines*; otherwise must be less than *ze_device_properties_t.numAsyncComputeEngines*. When using sub-devices the *ze_device_properties_t.numAsyncComputeEngines* must be queried from the sub-device being used.

19.5 Cmdlist

- Functions
  - *zeCommandListCreate*
  - *zeCommandListCreateImmediate*
  - *zeCommandListDestroy*
  - *zeCommandListClose*
  - *zeCommandListReset*
- Enumerations
  - *ze_command_list_desc_version_t*
  - *ze_command_list_flag_t*
- Structures
  - *ze_command_list_desc_t*

19.5.1 Cmdlist Functions

zeCommandListCreate

__ze_api_export *ze_result_t* __zecall **zeCommandListCreate**(ze_device_handle_t *hDevice*, **const** *ze_command_list_desc_t* **desc*, ze_command_list_handle_t **phCommandList*)

Creates a command list on the device for submitting commands to any command queue.

Parameters

- `hDevice`: handle of the device object
- `desc`: pointer to command list descriptor

  - `phCommandList`: pointer to handle of command list object created

- The command list can only be used on the device on which it was created.
- The command list is created in the 'open' state.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phCommandList`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_COMMAND_LIST_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeCommandListCreateImmediate

__ze_api_export *ze_result_t* __zecall **zeCommandListCreateImmediate**(ze_device_handle_t *hDevice*, **const** *ze_command_queue_desc_t* **altdesc*, ze_command_list_handle_t **phCommandList*)

Creates a command list on the device with an implicit command queue for immediate submission of commands.

Parameters

- `hDevice`: handle of the device object
- `altdesc`: pointer to command queue descriptor
- `phCommandList`: pointer to handle of command list object created

- The command list can only be used on the device on which it was created.
- The command list is created in the 'open' state and never needs to be closed.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == altdesc`
  - `nullptr == phCommandList`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *`ZE_COMMAND_QUEUE_DESC_VERSION_CURRENT`* `< altdesc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - altdesc->flags
  - altdesc->mode
  - altdesc->priority
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeCommandListDestroy

__ze_api_export *ze_result_t* __zecall **`zeCommandListDestroy`**(ze_command_list_handle_t *hCommandList*)

Destroys a command list.

Parameters

- `hCommandList`: [release] handle of command list object to destroy

- The application is responsible for making sure the device is not currently referencing the command list before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this command list.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`

- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeCommandListClose

__ze_api_export *ze_result_t* __zecall **`zeCommandListClose`**(ze_command_list_handle_t *hCommandList*)

Closes a command list; ready to be executed by a command queue.

Parameters

- `hCommandList`: handle of command list object to close

- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`

zeCommandListReset

__ze_api_export *ze_result_t* __zecall **`zeCommandListReset`**(ze_command_list_handle_t *hCommandList*)

Reset a command list to initial (empty) state; ready for appending commands.

Parameters

- `hCommandList`: handle of command list object to reset

- The application is responsible for making sure the device is not currently referencing the command list before it is reset
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`

19.5.2 Cmdlist Enums

ze_command_list_desc_version_t

enum ze_command_list_desc_version_t

API version of *ze_command_list_desc_t*.

Values:

enumerator ZE_COMMAND_LIST_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_command_list_flag_t

enum ze_command_list_flag_t

Supported command list creation flags.

Values:

enumerator ZE_COMMAND_LIST_FLAG_NONE = 0

default behavior

enumerator ZE_COMMAND_LIST_FLAG_COPY_ONLY = ZE_BIT(0)

command list **only** contains copy operations (and synchronization primitives). this command list may **only** be submitted to a command queue created with *ZE_COMMAND_QUEUE_FLAG_COPY_ONLY*.

enumerator ZE_COMMAND_LIST_FLAG_RELAXED_ORDERING = ZE_BIT(1)

driver may reorder programs and copys between barriers and synchronization primitives. using this flag may increase Host overhead of *zeCommandListClose*. therefore, this flag should **not** be set for low-latency usage-models.

enumerator ZE_COMMAND_LIST_FLAG_MAXIMIZE_THROUGHPUT = ZE_BIT(2)

driver may perform additional optimizations that increase dexecution throughput. using this flag may increase Host overhead of *zeCommandListClose* and *zeCommandQueueExecuteCommandLists*. therefore, this flag should **not** be set for low-latency usage-models.

enumerator ZE_COMMAND_LIST_FLAG_EXPLICIT_ONLY = ZE_BIT(3)

command list should be optimized for submission to a single command queue and device engine. driver **must** disable any implicit optimizations for distributing work across multiple engines. this flag should be used when applications want full control over multi-engine submission and scheduling.

19.5.3 Cmdlist Structures

ze_command_list_desc_t

struct ze_command_list_desc_t

Command List descriptor.

Public Members

ze_command_list_desc_version_t **version**
[in] *ZE_COMMAND_LIST_DESC_VERSION_CURRENT*

ze_command_list_flag_t **flags**
[in] creation flags

19.6 Barrier

- Functions
  - *zeCommandListAppendBarrier*
  - *zeCommandListAppendMemoryRangesBarrier*
  - *zeDeviceSystemBarrier*

19.6.1 Barrier Functions

zeCommandListAppendBarrier

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendBarrier**(ze_command_list_handle_t *hCommandList*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Appends an execution and global memory barrier into a command list.

Parameters

- `hCommandList`: handle of the command list
- `hSignalEvent`: [optional] handle of the event to signal on completion
- `numWaitEvents`: [optional] number of events to wait on before executing barrier
- `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before executing barrier

- If numWaitEvents is zero, then all previous commands are completed prior to the execution of the barrier.
- If numWaitEvents is non-zero, then then all phWaitEvents must be signaled prior to the execution of the barrier.
- This command blocks all following commands from beginning until the execution of the barrier completes.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkCmdPipelineBarrier**
- clEnqueueBarrierWithWaitList

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendMemoryRangesBarrier

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendMemoryRangesBarrier**(ze_command_list_handle_t *hCommandList*, uint32_t *numRanges*, **const** size_t **pRangeSizes*, **const** void ***pRanges*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Appends a global memory ranges barrier into a command list.

Parameters

- `hCommandList`: handle of the command list
- `numRanges`: number of memory ranges
- `pRangeSizes`: [range(0, numRanges)] array of sizes of memory range
- `pRanges`: [range(0, numRanges)] array of memory ranges
- `hSignalEvent`: [optional] handle of the event to signal on completion
- `numWaitEvents`: [optional] number of events to wait on before executing barrier
- `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before executing barrier

- If numWaitEvents is zero, then all previous commands are completed prior to the execution of the barrier.
- If numWaitEvents is non-zero, then then all phWaitEvents must be signaled prior to the execution of the barrier.
- This command blocks all following commands from beginning until the execution of the barrier completes.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pRangeSizes`
  - `nullptr == pRanges`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeDeviceSystemBarrier

__ze_api_export *ze_result_t* __zecall **zeDeviceSystemBarrier**(ze_device_handle_t *hDevice*)

Ensures in-bound writes to the device are globally observable.

Parameters

- `hDevice`: handle of the device

- This is a special-case system level barrier that can be used to ensure global observability of writes; typically needed after a producer (e.g., NIC) performs direct writes to the device's memory (e.g., Direct RDMA writes). This is typically required when the memory corresponding to the writes is subsequently accessed from a remote device.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`

19.7 Copy

- Functions
  - *zeCommandListAppendMemoryCopy*
  - *zeCommandListAppendMemoryFill*
  - *zeCommandListAppendMemoryCopyRegion*
  - *zeCommandListAppendImageCopy*
  - *zeCommandListAppendImageCopyRegion*
  - *zeCommandListAppendImageCopyToMemory*

  - *zeCommandListAppendImageCopyFromMemory*
  - *zeCommandListAppendMemoryPrefetch*
  - *zeCommandListAppendMemAdvise*
- Enumerations
  - *ze_memory_advice_t*
- Structures
  - *ze_copy_region_t*
  - *ze_image_region_t*

19.7.1 Copy Functions

zeCommandListAppendMemoryCopy

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendMemoryCopy** (ze_command_list_handle_t *hCommandList*, void **dstptr*, **const** void **srcptr*, size_t *size*, ze_event_handle_t *hEvent*)

Copies host, device, or shared memory.

Parameters

- `hCommandList`: handle of command list
- `dstptr`: pointer to destination memory to copy to
- `srcptr`: pointer to source memory to copy from
- `size`: size in bytes to copy
- `hEvent`: [optional] handle of the event to signal on completion

- The memory pointed to by both srcptr and dstptr must be accessible by the device on which the command list is created.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clEnqueueCopyBuffer**
- **clEnqueueReadBuffer**
- **clEnqueueWriteBuffer**
- **clEnqueueSVMMemcpy**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == dstptr`
  - `nullptr == srcptr`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendMemoryFill

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendMemoryFill`**(ze_command_list_handle_t *hCommandList*, void **ptr*, **`const`** void **pattern*, size_t *pattern_size*, size_t *size*, ze_event_handle_t *hEvent*)

Initializes host, device, or shared memory.

Parameters

- `hCommandList`: handle of command list
- `ptr`: pointer to memory to initialize
- `pattern`: pointer to value to initialize memory to
- `pattern_size`: size in bytes of the value to initialize memory to
- `size`: size in bytes to initialize
- `hEvent`: [optional] handle of the event to signal on completion

- The memory pointed to by dstptr must be accessible by the device on which the command list is created.
- The value to initialize memory to is described by the pattern and the pattern size.
- The pattern size must be a power of two.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clEnqueueFillBuffer**
- **clEnqueueSVMMemFill**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
  - `nullptr == pattern`

- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendMemoryCopyRegion

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendMemoryCopyRegion** (ze_command_list_handle_t *hCommandList*, void **dstptr*, **const** *ze_copy_region_t* **dstRegion*, uint32_t *dstPitch*, uint32_t *dstSlicePitch*, **const** void **srcptr*, **const** *ze_copy_region_t* **srcRegion*, uint32_t *srcPitch*, uint32_t *srcSlicePitch*, ze_event_handle_t *hEvent*)

Copies a region from a 2D or 3D array of host, device, or shared memory.

Parameters

- `hCommandList`: handle of command list
- `dstptr`: pointer to destination memory to copy to
- `dstRegion`: pointer to destination region to copy to
- `dstPitch`: destination pitch in bytes
- `dstSlicePitch`: destination slice pitch in bytes. This is required for 3D region copies where *ze_copy_region_t::depth* is not 0, otherwise it's ignored.
- `srcptr`: pointer to source memory to copy from
- `srcRegion`: pointer to source region to copy from
- `srcPitch`: source pitch in bytes
- `srcSlicePitch`: source slice pitch in bytes. This is required for 3D region copies where *ze_copy_region_t::depth* is not 0, otherwise it's ignored.
- `hEvent`: [optional] handle of the event to signal on completion

- The memory pointed to by both srcptr and dstptr must be accessible by the device on which the command list is created.
- The region width, height, and depth for both src and dst must be same. The origins can be different.
- The src and dst regions cannot be overlapping.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == dstptr`
  - `nullptr == dstRegion`
  - `nullptr == srcptr`
  - `nullptr == srcRegion`
- *ZE_RESULT_ERROR_OVERLAPPING_REGIONS*
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendImageCopy

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendImageCopy`**(ze_command_list_handle_t *hCommandList*, ze_image_handle_t *hDstImage*, ze_image_handle_t *hSrcImage*, ze_event_handle_t *hEvent*)

Copies a image.

Parameters

- `hCommandList`: handle of command list
- `hDstImage`: handle of destination image to copy to
- `hSrcImage`: handle of source image to copy from
- `hEvent`: [optional] handle of the event to signal on completion

- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clEnqueueCopyImage**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`

  - `nullptr == hDstImage`
  - `nullptr == hSrcImage`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendImageCopyRegion

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendImageCopyRegion**(ze_command_list_handle_t *hCommandList*, ze_image_handle_t *hDstImage*, ze_image_handle_t *hSrcImage*, **const** *ze_image_region_t *pDstRegion*, **const** *ze_image_region_t *pSrcRegion*, ze_event_handle_t *hEvent*)

Copies a region of a image to another image.

Parameters

- `hCommandList`: handle of command list
- `hDstImage`: handle of destination image to copy to
- `hSrcImage`: handle of source image to copy from
- `pDstRegion`: [optional] destination region descriptor
- `pSrcRegion`: [optional] source region descriptor
- `hEvent`: [optional] handle of the event to signal on completion

- The region width and height for both src and dst must be same. The origins can be different.
- The src and dst regions cannot be overlapping.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hDstImage`
  - `nullptr == hSrcImage`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

- *ZE_RESULT_ERROR_OVERLAPPING_REGIONS*

zeCommandListAppendImageCopyToMemory

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendImageCopyToMemory** (ze_command_list_handle_t *hCommandList*, void **dstptr*, ze_image_handle_t *hSrcImage*, **const** *ze_image_region_t* **pSrcRegion*, ze_event_handle_t *hEvent*)

Copies from a image to device or shared memory.

Parameters

- `hCommandList`: handle of command list
- `dstptr`: pointer to destination memory to copy to
- `hSrcImage`: handle of source image to copy from
- `pSrcRegion`: [optional] source region descriptor
- `hEvent`: [optional] handle of the event to signal on completion

- The memory pointed to by dstptr must be accessible by the device on which the command list is created.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clEnqueueReadImage

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hSrcImage`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == dstptr`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendImageCopyFromMemory

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendImageCopyFromMemory**(ze_command_list_handle_t *hCommandList*, ze_image_handle_t *hDstImage*, **const** void **srcptr*, **const** *ze_image_region_t* **pDstRegion*, ze_event_handle_t *hEvent*)

Copies to a image from device or shared memory.

Parameters

- `hCommandList`: handle of command list
- `hDstImage`: handle of destination image to copy to
- `srcptr`: pointer to source memory to copy from
- `pDstRegion`: [optional] destination region descriptor
- `hEvent`: [optional] handle of the event to signal on completion

- The memory pointed to by srcptr must be accessible by the device on which the command list is created.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clEnqueueWriteImage

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hDstImage`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == srcptr`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendMemoryPrefetch

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendMemoryPrefetch`**(ze_command_list_handle_t *hCommandList*, **`const`** void **ptr*, size_t *size*)

Asynchronously prefetches shared memory to the device associated with the specified command list.

Parameters

- `hCommandList`: handle of command list
- `ptr`: pointer to start of the memory range to prefetch
- `size`: size in bytes of the memory range to prefetch

- This is a hint to improve performance only and is not required for correctness.
- Only prefetching to the device associated with the specified command list is supported. Prefetching to the host or to a peer device is not supported.
- Prefetching may not be supported for all allocation types for all devices. If memory prefetching is not supported for the specified memory range the prefetch hint may be ignored.
- Prefetching may only be supported at a device-specific granularity, such as at a page boundary. In this case, the memory range may be expanded such that the start and end of the range satisfy granularity requirements.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clEnqueueSVMMigrateMem

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`

zeCommandListAppendMemAdvise

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendMemAdvise`**(ze_command_list_handle_t *hCommandList*, ze_device_handle_t *hDevice*, **`const`** void **ptr*, size_t *size*, *ze_memory_advice_t* *advice*)

Provides advice about the use of a shared memory range.

Parameters

- `hCommandList`: handle of command list
- `hDevice`: device associated with the memory advice
- `ptr`: Pointer to the start of the memory range
- `size`: Size in bytes of the memory range
- `advice`: Memory advice for the memory range

- Memory advice is a performance hint only and is not required for functional correctness.
- Memory advice can be used to override driver heuristics to explicitly control shared memory behavior.
- Not all memory advice hints may be supported for all allocation types for all devices. If a memory advice hint is not supported by the device it will be ignored.
- Memory advice may only be supported at a device-specific granularity, such as at a page boundary. In this case, the memory range may be expanded such that the start and end of the range satisfy granularity requirements.
- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - advice

19.7.2 Copy Enums

ze_memory_advice_t

enum ze_memory_advice_t

Supported memory advice hints.

Values:

enumerator ZE_MEMORY_ADVICE_SET_READ_MOSTLY = 0

hint that memory will be read from frequently and written to rarely

enumerator ZE_MEMORY_ADVICE_CLEAR_READ_MOSTLY

removes the affect of *ZE_MEMORY_ADVICE_SET_READ_MOSTLY*

enumerator ZE_MEMORY_ADVICE_SET_PREFERRED_LOCATION
hint that the preferred memory location is the specified device

enumerator ZE_MEMORY_ADVICE_CLEAR_PREFERRED_LOCATION
removes the affect of *ZE_MEMORY_ADVICE_SET_PREFERRED_LOCATION*

enumerator ZE_MEMORY_ADVICE_SET_ACCESSED_BY
hint that memory will be accessed by the specified device

enumerator ZE_MEMORY_ADVICE_CLEAR_ACCESSED_BY
removes the affect of *ZE_MEMORY_ADVICE_SET_ACCESSED_BY*

enumerator ZE_MEMORY_ADVICE_SET_NON_ATOMIC_MOSTLY
hints that memory will mostly be accessed non-atomically

enumerator ZE_MEMORY_ADVICE_CLEAR_NON_ATOMIC_MOSTLY
removes the affect of *ZE_MEMORY_ADVICE_SET_NON_ATOMIC_MOSTLY*

enumerator ZE_MEMORY_ADVICE_BIAS_CACHED
hints that memory should be cached

enumerator ZE_MEMORY_ADVICE_BIAS_UNCACHED
hints that memory should be not be cached

19.7.3 Copy Structures

ze_copy_region_t

struct ze_copy_region_t
Copy region descriptor.

Public Members

uint32_t **originX**
[in] The origin x offset for region in bytes

uint32_t **originY**
[in] The origin y offset for region in rows

uint32_t **originZ**
[in] The origin z offset for region in slices

uint32_t **width**
[in] The region width relative to origin in bytes

uint32_t **height**
[in] The region height relative to origin in rows

uint32_t **depth**
[in] The region depth relative to origin in slices. Set this to 0 for 2D copy.

ze_image_region_t

`struct ze_image_region_t`

Region descriptor.

Public Members

uint32_t **`originX`**

[in] The origin x offset for region in pixels

uint32_t **`originY`**

[in] The origin y offset for region in pixels

uint32_t **`originZ`**

[in] The origin z offset for region in pixels

uint32_t **`width`**

[in] The region width relative to origin in pixels

uint32_t **`height`**

[in] The region height relative to origin in pixels

uint32_t **`depth`**

[in] The region depth relative to origin. For 1D or 2D images, set this to 1.

19.8 Event

- Functions
  - *zeEventPoolCreate*
  - *zeEventPoolDestroy*
  - *zeEventCreate*
  - *zeEventDestroy*
  - *zeEventPoolGetIpcHandle*
  - *zeEventPoolOpenIpcHandle*
  - *zeEventPoolCloseIpcHandle*
  - *zeCommandListAppendSignalEvent*
  - *zeCommandListAppendWaitOnEvents*
  - *zeEventHostSignal*
  - *zeEventHostSynchronize*
  - *zeEventQueryStatus*
  - *zeCommandListAppendEventReset*
  - *zeEventHostReset*
  - *zeEventGetTimestamp*
- Enumerations
  - *ze_event_pool_desc_version_t*

  - *ze_event_pool_flag_t*
  - *ze_event_desc_version_t*
  - *ze_event_scope_flag_t*
  - *ze_event_timestamp_type_t*
- Structures
  - *ze_event_pool_desc_t*
  - *ze_event_desc_t*

19.8.1 Event Functions

zeEventPoolCreate

__ze_api_export *ze_result_t* __zecall **zeEventPoolCreate**(ze_driver_handle_t *hDriver*, **const** *ze_event_pool_desc_t* **desc*, uint32_t *numDevices*, ze_device_handle_t **phDevices*, ze_event_pool_handle_t **phEventPool*)

Creates a pool for a set of event(s) for the driver.

Parameters

- `hDriver`: handle of the driver instance
- `desc`: pointer to event pool descriptor
- `numDevices`: number of device handles
- `phDevices`: [optional][range(0, numDevices)] array of device handles which have visibility to the event pool. if nullptr, then event pool is visible to all devices supported by the driver instance.
- `phEventPool`: pointer handle of event pool object created

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phEventPool`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *`ZE_EVENT_POOL_DESC_VERSION_CURRENT`* `< desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*

    - desc->flags
  - *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
  - *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeEventPoolDestroy

__ze_api_export *ze_result_t* __zecall **`zeEventPoolDestroy`** (ze_event_pool_handle_t *hEventPool*)

Deletes an event pool object.

Parameters

- `hEventPool`: [release] handle of event pool object to destroy

- The application is responsible for destroying all event handles created from the pool before destroying the pool itself
- The application is responsible for making sure the device is not currently referencing the any event within the pool before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this event pool
- The application may **not** call this function from simultaneous threads with the same event pool handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEventPool`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeEventCreate

__ze_api_export *ze_result_t* __zecall **`zeEventCreate`** (ze_event_pool_handle_t *hEventPool*, **`const`** *ze_event_desc_t* **desc*, ze_event_handle_t **phEvent*)

Creates an event on the device.

Parameters

- `hEventPool`: handle of the event pool
- `desc`: pointer to event descriptor
- `phEvent`: pointer to handle of event object created

- Multiple events cannot be created using the same location within the same pool.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clCreateUserEvent**
- vkCreateEvent

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEventPool`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phEvent`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *ZE_EVENT_DESC_VERSION_CURRENT* `< desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->signal
  - desc->wait
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*

zeEventDestroy

__ze_api_export *ze_result_t* __zecall **zeEventDestroy**(ze_event_handle_t *hEvent*)
Deletes an event object.

Parameters

- `hEvent`: [release] handle of event object to destroy

- The application is responsible for making sure the device is not currently referencing the event before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this event
- The application may **not** call this function from simultaneous threads with the same event handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clReleaseEvent**
- vkDestroyEvent

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeEventPoolGetIpcHandle

__ze_api_export *ze_result_t* __zecall **zeEventPoolGetIpcHandle**(ze_event_pool_handle_t *hEventPool*, *ze_ipc_event_pool_handle_t* **phIpc*)

Gets an IPC event pool handle for the specified event handle that can be shared with another process.

Parameters

- `hEventPool`: handle of event pool object
- `phIpc`: Returned IPC event handle

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEventPool`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phIpc`

zeEventPoolOpenIpcHandle

__ze_api_export *ze_result_t* __zecall **zeEventPoolOpenIpcHandle**(ze_driver_handle_t *hDriver*, *ze_ipc_event_pool_handle_t* *hIpc*, ze_event_pool_handle_t **phEventPool*)

Opens an IPC event pool handle to retrieve an event pool handle from another process.

Parameters

- `hDriver`: handle of the driver to associate with the IPC event pool handle
- `hIpc`: IPC event handle
- `phEventPool`: pointer handle of event pool object created

- The event handle in this process should not be freed with *zeEventPoolDestroy*, but rather with *zeEventPoolCloseIpcHandle*.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phEventPool`

zeEventPoolCloseIpcHandle

__ze_api_export *ze_result_t* __zecall **zeEventPoolCloseIpcHandle** (ze_event_pool_handle_t *hEventPool*)

Closes an IPC event handle in the current process.

Parameters

- `hEventPool`: [release] handle of event pool object

- Closes an IPC event handle by destroying events that were opened in this process using *zeEventPoolOpenIpcHandle*.
- The application may **not** call this function from simultaneous threads with the same event pool handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEventPool`

zeCommandListAppendSignalEvent

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendSignalEvent** (ze_command_list_handle_t *hCommandList*, ze_event_handle_t *hEvent*)

Appends a signal of the event from the device into a command list.

Parameters

- `hCommandList`: handle of the command list
- `hEvent`: handle of the event

- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clSetUserEventStatus**

- vkCmdSetEvent

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendWaitOnEvents

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendWaitOnEvents** (ze_command_list_handle_t *hCommandList*, uint32_t *numEvents*, ze_event_handle_t **phEvents*)

Appends wait on event(s) on the device into a command list.

Parameters

- `hCommandList`: handle of the command list
- `numEvents`: number of events to wait on before continuing
- `phEvents`: [range(0, numEvents)] handle of the events to wait on before continuing

- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phEvents`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeEventHostSignal

__ze_api_export *ze_result_t* __zecall **zeEventHostSignal**(ze_event_handle_t *hEvent*)

Signals a event from host.

Parameters

- `hEvent`: handle of the event

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clSetUserEventStatus

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeEventHostSynchronize

__ze_api_export *ze_result_t* __zecall **zeEventHostSynchronize**(ze_event_handle_t *hEvent*, uint32_t *timeout*)

The current host thread waits on an event to be signaled.

Parameters

- `hEvent`: handle of the event
- `timeout`: if non-zero, then indicates the maximum time (in nanoseconds) to yield before returning *ZE_RESULT_SUCCESS* or *ZE_RESULT_NOT_READY*; if zero, then operates exactly like *zeEventQueryStatus*; if UINT32_MAX, then function will not return until complete or device is lost.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clWaitForEvents

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

    - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*
- *ZE_RESULT_NOT_READY*
    - timeout expired

zeEventQueryStatus

__ze_api_export *ze_result_t* __zecall **zeEventQueryStatus** (ze_event_handle_t *hEvent*)

Queries an event object's status.

Parameters

- `hEvent`: handle of the event

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clGetEventInfo**
- vkGetEventStatus

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*
- *ZE_RESULT_NOT_READY*
    - not signaled

zeCommandListAppendEventReset

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendEventReset** (ze_command_list_handle_t *hCommandList*, ze_event_handle_t *hEvent*)

Reset an event back to not signaled state.

Parameters

- `hCommandList`: handle of the command list
- `hEvent`: handle of the event

- The application may **not** call this function from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- vkResetEvent

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeEventHostReset

__ze_api_export *ze_result_t* __zecall **zeEventHostReset**(ze_event_handle_t *hEvent*)

Reset an event back to not signaled state.

Parameters

- `hEvent`: handle of the event

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- vkResetEvent

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeEventGetTimestamp

__ze_api_export *ze_result_t* __zecall **zeEventGetTimestamp**(ze_event_handle_t *hEvent*, *ze_event_timestamp_type_t* *timestampType*, void **dstptr*)

Query timestamp information associated with an event. Event must come from an event pool that was created using *ZE_EVENT_POOL_FLAG_TIMESTAMP* flag.

Parameters

- `hEvent`: handle of the event

    - `timestampType`: specifies timestamp type to query for that is associated with hEvent.
    - `dstptr`: pointer to memory for where timestamp will be written to. The size of timestamp is specified in the *ze_event_timestamp_type_t* definition.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - timestampType
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == dstptr`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

19.8.2 Event Enums

ze_event_pool_desc_version_t

enum ze_event_pool_desc_version_t

API version of *ze_event_pool_desc_t*.

Values:

enumerator ZE_EVENT_POOL_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_event_pool_flag_t

enum ze_event_pool_flag_t

Supported event pool creation flags.

Values:

enumerator ZE_EVENT_POOL_FLAG_DEFAULT = 0

signals and waits visible to the entire device and peer devices

enumerator ZE_EVENT_POOL_FLAG_HOST_VISIBLE = ZE_BIT(0)

signals and waits are also visible to host

enumerator ZE_EVENT_POOL_FLAG_IPC = ZE_BIT(1)

signals and waits may be shared across processes

enumerator ZE_EVENT_POOL_FLAG_TIMESTAMP = ZE_BIT(2)

Indicates all events in pool will contain timestamp information that can be queried using *zeEventGetTimestamp*

ze_event_desc_version_t

enum ze_event_desc_version_t

API version of *ze_event_desc_t*.

Values:

enumerator ZE_EVENT_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_event_scope_flag_t

enum ze_event_scope_flag_t

Supported event scope flags.

Values:

enumerator ZE_EVENT_SCOPE_FLAG_NONE = 0
execution synchronization only; no cache hierarchies are flushed or invalidated

enumerator ZE_EVENT_SCOPE_FLAG_SUBDEVICE = ZE_BIT(0)
cache hierarchies are flushed or invalidated sufficient for local sub-device access

enumerator ZE_EVENT_SCOPE_FLAG_DEVICE = ZE_BIT(1)
cache hierarchies are flushed or invalidated sufficient for global device access and peer device access

enumerator ZE_EVENT_SCOPE_FLAG_HOST = ZE_BIT(2)
cache hierarchies are flushed or invalidated sufficient for device and host access

ze_event_timestamp_type_t

enum ze_event_timestamp_type_t

Supported timestamp types.

Values:

enumerator ZE_EVENT_TIMESTAMP_GLOBAL_START = 0
wall-clock time start in GPU clocks for event. Data is uint64_t.

enumerator ZE_EVENT_TIMESTAMP_GLOBAL_END
wall-clock time end in GPU clocks for event.Data is uint64_t.

enumerator ZE_EVENT_TIMESTAMP_CONTEXT_START
context time start in GPU clocks for event. Only includes time while HW context is actively running on GPU. Data is uint64_t.

enumerator ZE_EVENT_TIMESTAMP_CONTEXT_END
context time end in GPU clocks for event. Only includes time while HW context is actively running on GPU. Data is uint64_t.

19.8.3 Event Structures

ze_event_pool_desc_t

struct ze_event_pool_desc_t
Event pool descriptor.

Public Members

ze_event_pool_desc_version_t **version**
[in] *ZE_EVENT_POOL_DESC_VERSION_CURRENT*

ze_event_pool_flag_t **flags**
[in] creation flags

uint32_t **count**
[in] number of events within the pool

ze_event_desc_t

struct ze_event_desc_t
Event descriptor.

Public Members

ze_event_desc_version_t **version**
[in] *ZE_EVENT_DESC_VERSION_CURRENT*

uint32_t **index**
[in] index of the event within the pool; must be less-than the count specified during pool creation

ze_event_scope_flag_t **signal**
[in] defines the scope of relevant cache hierarchies to flush on a signal action before the event is triggered

ze_event_scope_flag_t **wait**
[in] defines the scope of relevant cache hierarchies to invalidate on a wait action after the event is complete

19.9 Fence

- Functions
  - *zeFenceCreate*
  - *zeFenceDestroy*
  - *zeFenceHostSynchronize*
  - *zeFenceQueryStatus*
  - *zeFenceReset*
- Enumerations
  - *ze_fence_desc_version_t*
  - *ze_fence_flag_t*

- Structures
  - *ze_fence_desc_t*

19.9.1 Fence Functions

zeFenceCreate

__ze_api_export *ze_result_t* __zecall **zeFenceCreate**(ze_command_queue_handle_t *hCommandQueue*, **const** *ze_fence_desc_t* **desc*, ze_fence_handle_t **phFence*)

Creates a fence object on the device's command queue.

Parameters

- `hCommandQueue`: handle of command queue
- `desc`: pointer to fence descriptor
- `phFence`: pointer to handle of fence object created

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkCreateFence**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandQueue`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phFence`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_FENCE_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeFenceDestroy

__ze_api_export *ze_result_t* __zecall **`zeFenceDestroy`**(ze_fence_handle_t *hFence*)

Deletes a fence object.

Parameters

- `hFence`: [release] handle of fence object to destroy

- The application is responsible for making sure the device is not currently referencing the fence before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this fence
- The application may **not** call this function from simultaneous threads with the same fence handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkDestroyFence**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFence`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeFenceHostSynchronize

__ze_api_export *ze_result_t* __zecall **`zeFenceHostSynchronize`**(ze_fence_handle_t *hFence*, uint32_t *timeout*)

The current host thread waits on a fence to be signaled.

Parameters

- `hFence`: handle of the fence
- `timeout`: if non-zero, then indicates the maximum time (in nanoseconds) to yield before returning *ZE_RESULT_SUCCESS* or *ZE_RESULT_NOT_READY*; if zero, then operates exactly like *zeFenceQueryStatus*; if UINT32_MAX, then function will not return until complete or device is lost.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkWaitForFences**

Return

- *ZE_RESULT_SUCCESS*

- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFence`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*
- *ZE_RESULT_NOT_READY*
  - timeout expired

zeFenceQueryStatus

__ze_api_export *ze_result_t* __zecall **`zeFenceQueryStatus`** (ze_fence_handle_t *hFence*)

Queries a fence object's status.

Parameters

- `hFence`: handle of the fence

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkGetFenceStatus**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFence`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*
- *ZE_RESULT_NOT_READY*
  - not signaled

zeFenceReset

__ze_api_export *ze_result_t* __zecall **`zeFenceReset`** (ze_fence_handle_t *hFence*)

Reset a fence back to the not signaled state.

Parameters

- `hFence`: handle of the fence

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **vkResetFences**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFence`

19.9.2 Fence Enums

ze_fence_desc_version_t

`enum ze_fence_desc_version_t`

API version of *ze_fence_desc_t*.

Values:

`enumerator ZE_FENCE_DESC_VERSION_CURRENT` = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_fence_flag_t

`enum ze_fence_flag_t`

Supported fence creation flags.

Values:

`enumerator ZE_FENCE_FLAG_NONE` = 0

default behavior

19.9.3 Fence Structures

ze_fence_desc_t

`struct ze_fence_desc_t`

Fence descriptor.

Public Members

ze_fence_desc_version_t **`version`**

[in] *ZE_FENCE_DESC_VERSION_CURRENT*

ze_fence_flag_t **`flags`**

[in] creation flags

19.10 Image

- Functions
  - *zeImageGetProperties*
  - *zeImageCreate*
  - *zeImageDestroy*
- Enumerations
  - *ze_image_desc_version_t*
  - *ze_image_flag_t*
  - *ze_image_type_t*
  - *ze_image_format_layout_t*
  - *ze_image_format_type_t*
  - *ze_image_format_swizzle_t*
  - *ze_image_properties_version_t*
  - *ze_image_sampler_filter_flags_t*
- Structures
  - *ze_image_format_desc_t*
  - *ze_image_desc_t*
  - *ze_image_properties_t*

19.10.1 Image Functions

zeImageGetProperties

__ze_api_export *ze_result_t* __zecall **zeImageGetProperties**(ze_device_handle_t *hDevice*, **const** *ze_image_desc_t* **desc*, *ze_image_properties_t* **pImageProperties*)

Retrieves supported properties of an image.

Parameters

- `hDevice`: handle of the device
- `desc`: pointer to image descriptor
- `pImageProperties`: pointer to image properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == pImageProperties`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_IMAGE_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
  - desc->type

zeImageCreate

__ze_api_export *ze_result_t* __zecall **zeImageCreate**(ze_device_handle_t *hDevice*, **const** *ze_image_desc_t* **desc*, ze_image_handle_t **phImage*)

Creates a image object on the device.

Parameters

- `hDevice`: handle of the device
- `desc`: pointer to image descriptor
- `phImage`: pointer to handle of image object created

- The image is only visible to the device on which it was created.
- The image can be copied to another device using the ::*zeCommandListAppendImageCopy* functions.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Remark *Analogues*

- clCreateImage

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phImage`

- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_IMAGE_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
  - desc->type
- *ZE_RESULT_ERROR_UNSUPPORTED_IMAGE_FORMAT*
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeImageDestroy

__ze_api_export *ze_result_t* __zecall **zeImageDestroy**(ze_image_handle_t *hImage*)
Deletes a image object.

Parameters

- `hImage`: [release] handle of image object to destroy

- The application is responsible for making sure the device is not currently referencing the image before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this image
- The application may **not** call this function from simultaneous threads with the same image handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hImage`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

19.10.2 Image Enums

ze_image_desc_version_t

enum ze_image_desc_version_t
API version of *ze_image_desc_t*.

Values:

enumerator ZE_IMAGE_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_image_flag_t

enum ze_image_flag_t

Supported image creation flags.

Values:

enumerator ZE_IMAGE_FLAG_PROGRAM_READ = ZE_BIT(0)
programs will read contents

enumerator ZE_IMAGE_FLAG_PROGRAM_WRITE = ZE_BIT(1)
programs will write contents

enumerator ZE_IMAGE_FLAG_BIAS_CACHED = ZE_BIT(2)
device should cache contents

enumerator ZE_IMAGE_FLAG_BIAS_UNCACHED = ZE_BIT(3)
device should not cache contents

ze_image_type_t

enum ze_image_type_t

Supported image types.

Values:

enumerator ZE_IMAGE_TYPE_1D
1D

enumerator ZE_IMAGE_TYPE_1DARRAY
1D array

enumerator ZE_IMAGE_TYPE_2D
2D

enumerator ZE_IMAGE_TYPE_2DARRAY
2D array

enumerator ZE_IMAGE_TYPE_3D
3D

enumerator ZE_IMAGE_TYPE_BUFFER
Buffer.

ze_image_format_layout_t

enum ze_image_format_layout_t

Supported image format layouts.

Values:

enumerator ZE_IMAGE_FORMAT_LAYOUT_8
8-bit single component layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_16
16-bit single component layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_32
32-bit single component layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_8_8
2-component 8-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_8_8_8_8
4-component 8-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_16_16
2-component 16-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_16_16_16_16
4-component 16-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_32_32
2-component 32-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_32_32_32_32
4-component 32-bit layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_10_10_10_2
4-component 10_10_10_2 layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_11_11_10
3-component 11_11_10 layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_5_6_5
3-component 5_6_5 layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_5_5_5_1
4-component 5_5_5_1 layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_4_4_4_4
4-component 4_4_4_4 layout

enumerator ZE_IMAGE_FORMAT_LAYOUT_Y8
Media Format: Y8. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_NV12
Media Format: NV12. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_YUYV
Media Format: YUYV. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_VYUY
Media Format: VYUY. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_YVYU
Media Format: YVYU. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_UYVY
Media Format: UYVY. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_AYUV
Media Format: AYUV. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_P010
Media Format: P010. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_Y410
Media Format: Y410. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_P012
Media Format: P012. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_Y16
Media Format: Y16. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_P016
Media Format: P016. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_Y216
Media Format: Y216. Format type and swizzle is ignored for this.

enumerator ZE_IMAGE_FORMAT_LAYOUT_P216
Media Format: P216. Format type and swizzle is ignored for this.

ze_image_format_type_t

enum ze_image_format_type_t
Supported image format types.

Values:

enumerator ZE_IMAGE_FORMAT_TYPE_UINT
Unsigned integer.

enumerator ZE_IMAGE_FORMAT_TYPE_SINT
Signed integer.

enumerator ZE_IMAGE_FORMAT_TYPE_UNORM
Unsigned normalized integer.

enumerator ZE_IMAGE_FORMAT_TYPE_SNORM
Signed normalized integer.

enumerator ZE_IMAGE_FORMAT_TYPE_FLOAT
Float.

ze_image_format_swizzle_t

enum ze_image_format_swizzle_t
Supported image format component swizzle into channel.

Values:

enumerator ZE_IMAGE_FORMAT_SWIZZLE_R
Red component.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_G
Green component.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_B
Blue component.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_A
Alpha component.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_0
Zero.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_1
One.

enumerator ZE_IMAGE_FORMAT_SWIZZLE_X
Don’t care.

ze_image_properties_version_t

enum ze_image_properties_version_t
API version of *ze_image_properties_t*.

Values:

enumerator ZE_IMAGE_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_image_sampler_filter_flags_t

enum ze_image_sampler_filter_flags_t
Supported sampler filtering flags.

Values:

enumerator ZE_IMAGE_SAMPLER_FILTER_FLAGS_NONE = 0
device does not support filtering

enumerator ZE_IMAGE_SAMPLER_FILTER_FLAGS_POINT = ZE_BIT(0)
device supports point filtering

enumerator ZE_IMAGE_SAMPLER_FILTER_FLAGS_LINEAR = ZE_BIT(1)
device supports linear filtering

19.10.3 Image Structures

ze_image_format_desc_t

struct ze_image_format_desc_t
Image format descriptor.

Public Members

ze_image_format_layout_t **layout**
[in] image format component layout

ze_image_format_type_t **type**
[in] image format type. Media formats can’t be used for *ZE_IMAGE_TYPE_BUFFER*.

ze_image_format_swizzle_t **x**
[in] image component swizzle into channel x

ze_image_format_swizzle_t **y**
[in] image component swizzle into channel y

ze_image_format_swizzle_t **z**
[in] image component swizzle into channel z

ze_image_format_swizzle_t **w**
[in] image component swizzle into channel w

ze_image_desc_t

struct ze_image_desc_t
Image descriptor.

Public Members

ze_image_desc_version_t **version**
[in] *ZE_IMAGE_DESC_VERSION_CURRENT*

ze_image_flag_t **flags**
[in] creation flags

ze_image_type_t **type**
[in] image type

ze_image_format_desc_t **format**
[in] image format

uint64_t **width**
[in] width in pixels for *ze_image_type_t*::1D/2D/3D and bytes for Buffer, see *ze_device_image_properties_t::maxImageDims1D*/2D/3D and maxImageBufferSize.

uint32_t **height**
[in] height in pixels (2D or 3D only), see *ze_device_image_properties_t::maxImageDims2D*/3D

uint32_t **depth**
[in] depth in pixels (3D only), see *ze_device_image_properties_t::maxImageDims3D*

uint32_t **arraylevels**
[in] array levels (array types only), see *ze_device_image_properties_t::maxImageArraySlices*

uint32_t **miplevels**
[in] mipmap levels (must be 0)

ze_image_properties_t

struct ze_image_properties_t
Image properties.

Public Members

ze_image_properties_version_t **version**
[in] *ZE_IMAGE_PROPERTIES_VERSION_CURRENT*

ze_image_sampler_filter_flags_t **samplerFilterFlags**
[out] supported sampler filtering

19.11 Memory

- Functions
  - *zeDriverAllocSharedMem*
  - *zeDriverAllocDeviceMem*
  - *zeDriverAllocHostMem*
  - *zeDriverFreeMem*
  - *zeDriverGetMemAllocProperties*
  - *zeDriverGetMemAddressRange*
  - *zeDriverGetMemIpcHandle*
  - *zeDriverOpenMemIpcHandle*
  - *zeDriverCloseMemIpcHandle*
- Enumerations
  - *ze_device_mem_alloc_desc_version_t*
  - *ze_device_mem_alloc_flag_t*
  - *ze_host_mem_alloc_desc_version_t*
  - *ze_host_mem_alloc_flag_t*
  - *ze_memory_allocation_properties_version_t*
  - *ze_memory_type_t*
  - *ze_ipc_memory_flag_t*
- Structures
  - *ze_device_mem_alloc_desc_t*
  - *ze_host_mem_alloc_desc_t*
  - *ze_memory_allocation_properties_t*

19.11.1 Memory Functions

zeDriverAllocSharedMem

__ze_api_export *ze_result_t* __zecall **zeDriverAllocSharedMem**(ze_driver_handle_t *hDriver*, **const** *ze_device_mem_alloc_desc_t* **device_desc*, **const** *ze_host_mem_alloc_desc_t* **host_desc*, size_t *size*, size_t *alignment*, ze_device_handle_t *hDevice*, void ***pptr*)

Allocates memory that is shared between the host and one or more devices.

Parameters

- `hDriver`: handle of the driver instance
- `device_desc`: pointer to device mem alloc descriptor

  - `host_desc`: pointer to host mem alloc descriptor
  - `size`: size in bytes to allocate
  - `alignment`: minimum alignment in bytes for the allocation
  - `hDevice`: [optional] device handle to associate with
  - `pptr`: pointer to shared allocation

- Shared allocations share ownership between the host and one or more devices.
- Shared allocations may optionally be associated with a device by passing a handle to the device.
- Devices supporting only single-device shared access capabilities may access shared memory associated with the device. For these devices, ownership of the allocation is shared between the host and the associated device only.
- Passing nullptr as the device handle does not associate the shared allocation with any device. For allocations with no associated device, ownership of the allocation is shared between the host and all devices supporting cross-device shared access capabilities.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == device_desc`
  - `nullptr == host_desc`
  - `nullptr == pptr`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_DEVICE_MEM_ALLOC_DESC_VERSION_CURRENT < device_desc->version`
  - `ZE_HOST_MEM_ALLOC_DESC_VERSION_CURRENT < host_desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - device_desc->flags
  - host_desc->flags
- *ZE_RESULT_ERROR_UNSUPPORTED_SIZE*
- *ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT*
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDriverAllocDeviceMem

__ze_api_export *ze_result_t* __zecall **`zeDriverAllocDeviceMem`**(ze_driver_handle_t *hDriver*, **`const`** *ze_device_mem_alloc_desc_t* **device_desc*, size_t *size*, size_t *alignment*, ze_device_handle_t *hDevice*, void ***pptr*)

Allocates memory specific to a device.

Parameters

- `hDriver`: handle of the driver instance
- `device_desc`: pointer to device mem alloc descriptor
- `size`: size in bytes to allocate
- `alignment`: minimum alignment in bytes for the allocation
- `hDevice`: handle of the device
- `pptr`: pointer to device allocation

- A device allocation is owned by a specific device.
- In general, a device allocation may only be accessed by the device that owns it.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == device_desc`
  - `nullptr == pptr`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_DEVICE_MEM_ALLOC_DESC_VERSION_CURRENT < device_desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - device_desc->flags
- *ZE_RESULT_ERROR_UNSUPPORTED_SIZE*
- *ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT*
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDriverAllocHostMem

__ze_api_export *ze_result_t* __zecall **zeDriverAllocHostMem**(ze_driver_handle_t *hDriver*, **const** *ze_host_mem_alloc_desc_t* **host_desc*, size_t *size*, size_t *alignment*, void ***pptr*)

Allocates host memory.

Parameters

- `hDriver`: handle of the driver instance
- `host_desc`: pointer to host mem alloc descriptor
- `size`: size in bytes to allocate
- `alignment`: minimum alignment in bytes for the allocation
- `pptr`: pointer to host allocation

- A host allocation is owned by the host process.
- Host allocations are accessible by the host and all devices within the driver driver.
- Host allocations are frequently used as staging areas to transfer data to or from devices.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == host_desc`
  - `nullptr == pptr`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *ZE_HOST_MEM_ALLOC_DESC_VERSION_CURRENT* `< host_desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - host_desc->flags
- *ZE_RESULT_ERROR_UNSUPPORTED_SIZE*
- *ZE_RESULT_ERROR_UNSUPPORTED_ALIGNMENT*
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDriverFreeMem

__ze_api_export *ze_result_t* __zecall **zeDriverFreeMem**(ze_driver_handle_t *hDriver*, void **ptr*)

Frees allocated host memory, device memory, or shared memory.

Parameters

- `hDriver`: handle of the driver instance
- `ptr`: [release] pointer to memory to free

- The application is responsible for making sure the device is not currently referencing the memory before it is freed
- The implementation of this function will immediately free all Host and Device allocations associated with this memory
- The application may **not** call this function from simultaneous threads with the same pointer.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`

zeDriverGetMemAllocProperties

__ze_api_export *ze_result_t* __zecall **zeDriverGetMemAllocProperties**(ze_driver_handle_t *hDriver*, **const** void **ptr*, *ze_memory_allocation_properties_t* **pMemAllocProperties*, ze_device_handle_t **phDevice*)

Retrieves attributes of a memory allocation.

Parameters

- `hDriver`: handle of the driver instance
- `ptr`: memory pointer to query
- `pMemAllocProperties`: query result for memory allocation properties
- `phDevice`: [optional] device associated with this allocation

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
  - `nullptr == pMemAllocProperties`

zeDriverGetMemAddressRange

__ze_api_export *ze_result_t* __zecall **zeDriverGetMemAddressRange**(ze_driver_handle_t *hDriver*, **const** void **ptr*, void ***pBase*, size_t **pSize*)

Retrieves the base address and/or size of an allocation.

Parameters

- `hDriver`: handle of the driver instance
- `ptr`: memory pointer to query
- `pBase`: [optional] base address of the allocation
- `pSize`: [optional] size of the allocation

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`

zeDriverGetMemIpcHandle

__ze_api_export *ze_result_t* __zecall **zeDriverGetMemIpcHandle**(ze_driver_handle_t *hDriver*, **const** void **ptr*, *ze_ipc_mem_handle_t* **pIpcHandle*)

Creates an IPC memory handle for the specified allocation in the sending process.

Parameters

- `hDriver`: handle of the driver instance
- `ptr`: pointer to the device memory allocation
- `pIpcHandle`: Returned IPC memory handle

- Takes a pointer to the base of a device memory allocation and exports it for use in another process.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
  - `nullptr == pIpcHandle`

zeDriverOpenMemIpcHandle

__ze_api_export *ze_result_t* __zecall **zeDriverOpenMemIpcHandle** (ze_driver_handle_t *hDriver*, ze_device_handle_t *hDevice*, *ze_ipc_mem_handle_t* *handle*, *ze_ipc_memory_flag_t* *flags*, void ***pptr*)

Opens an IPC memory handle to retrieve a device pointer in a receiving process.

Parameters

- `hDriver`: handle of the driver instance
- `hDevice`: handle of the device to associate with the IPC memory handle
- `handle`: IPC memory handle
- `flags`: flags controlling the operation
- `pptr`: pointer to device allocation in this process

- Takes an IPC memory handle from a sending process and associates it with a device pointer usable in this process.
- The device pointer in this process should not be freed with *zeDriverFreeMem*, but rather with *zeDriver-CloseMemIpcHandle*.
- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
  - `nullptr == hDevice`

- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - flags
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pptr`

zeDriverCloseMemIpcHandle

__ze_api_export *ze_result_t* __zecall **zeDriverCloseMemIpcHandle** (ze_driver_handle_t *hDriver*, **const** void **ptr*)

Closes an IPC memory handle in a receiving process.

Parameters

- `hDriver`: handle of the driver instance
- `ptr`: [release] pointer to device allocation in this process

- Closes an IPC memory handle by unmapping memory that was opened in this process using *zeDriverOpenMemIpcHandle*.
- The application may **not** call this function from simultaneous threads with the same pointer.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`

19.11.2 Memory Enums

ze_device_mem_alloc_desc_version_t

enum ze_device_mem_alloc_desc_version_t

API version of *ze_device_mem_alloc_desc_t*.

Values:

enumerator ZE_DEVICE_MEM_ALLOC_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_device_mem_alloc_flag_t

enum ze_device_mem_alloc_flag_t

Supported memory allocation flags.

Values:

enumerator ZE_DEVICE_MEM_ALLOC_FLAG_DEFAULT = 0
implicit default behavior; uses driver-based heuristics

enumerator ZE_DEVICE_MEM_ALLOC_FLAG_BIAS_CACHED = ZE_BIT(0)
device should cache allocation

enumerator ZE_DEVICE_MEM_ALLOC_FLAG_BIAS_UNCACHED = ZE_BIT(1)
device should not cache allocation (UC)

ze_host_mem_alloc_desc_version_t

enum ze_host_mem_alloc_desc_version_t

API version of *ze_host_mem_alloc_desc_t*.

Values:

enumerator ZE_HOST_MEM_ALLOC_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_host_mem_alloc_flag_t

enum ze_host_mem_alloc_flag_t

Supported host memory allocation flags.

Values:

enumerator ZE_HOST_MEM_ALLOC_FLAG_DEFAULT = 0
implicit default behavior; uses driver-based heuristics

enumerator ZE_HOST_MEM_ALLOC_FLAG_BIAS_CACHED = ZE_BIT(0)
host should cache allocation

enumerator ZE_HOST_MEM_ALLOC_FLAG_BIAS_UNCACHED = ZE_BIT(1)
host should not cache allocation (UC)

enumerator ZE_HOST_MEM_ALLOC_FLAG_BIAS_WRITE_COMBINED = ZE_BIT(2)
host memory should be allocated write-combined (WC)

ze_memory_allocation_properties_version_t

enum ze_memory_allocation_properties_version_t

API version of *ze_memory_allocation_properties_t*.

Values:

enumerator ZE_MEMORY_ALLOCATION_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_memory_type_t

enum ze_memory_type_t
Memory allocation type.

Values:

enumerator ZE_MEMORY_TYPE_UNKNOWN = 0
the memory pointed to is of unknown type

enumerator ZE_MEMORY_TYPE_HOST
the memory pointed to is a host allocation

enumerator ZE_MEMORY_TYPE_DEVICE
the memory pointed to is a device allocation

enumerator ZE_MEMORY_TYPE_SHARED
the memory pointed to is a shared ownership allocation

ze_ipc_memory_flag_t

enum ze_ipc_memory_flag_t
Supported IPC memory flags.

Values:

enumerator ZE_IPC_MEMORY_FLAG_NONE = 0
No special flags.

19.11.3 Memory Structures

ze_device_mem_alloc_desc_t

struct ze_device_mem_alloc_desc_t
Device mem alloc descriptor.

Public Members

ze_device_mem_alloc_desc_version_t **version**
[in] *ZE_DEVICE_MEM_ALLOC_DESC_VERSION_CURRENT*

ze_device_mem_alloc_flag_t **flags**
[in] flags specifying additional allocation controls

uint32_t **ordinal**
[in] ordinal of the device's local memory to allocate from; must be less than the count returned from *zeDeviceGetMemoryProperties*

ze_host_mem_alloc_desc_t

struct ze_host_mem_alloc_desc_t
Host mem alloc descriptor.

Public Members

ze_host_mem_alloc_desc_version_t **version**
[in] *ZE_HOST_MEM_ALLOC_DESC_VERSION_CURRENT*

ze_host_mem_alloc_flag_t **flags**
[in] flags specifying additional allocation controls

ze_memory_allocation_properties_t

struct ze_memory_allocation_properties_t
Memory allocation properties queried using *zeDriverGetMemAllocProperties*.

Public Members

ze_memory_allocation_properties_version_t **version**
[in] *ZE_MEMORY_ALLOCATION_PROPERTIES_VERSION_CURRENT*

ze_memory_type_t **type**
[out] type of allocated memory

uint64_t **id**
[out] identifier for this allocation

19.12 Module

- Functions
  - *zeModuleCreate*
  - *zeModuleDestroy*
  - *zeModuleBuildLogDestroy*
  - *zeModuleBuildLogGetString*
  - *zeModuleGetNativeBinary*
  - *zeModuleGetGlobalPointer*
  - *zeModuleGetKernelNames*
  - *zeKernelCreate*
  - *zeKernelDestroy*
  - *zeModuleGetFunctionPointer*
  - *zeKernelSetGroupSize*
  - *zeKernelSuggestGroupSize*
  - *zeKernelSuggestMaxCooperativeGroupCount*

  - *zeKernelSetArgumentValue*
  - *zeKernelSetAttribute*
  - *zeKernelGetAttribute*
  - *zeKernelSetIntermediateCacheConfig*
  - *zeKernelGetProperties*
  - *zeCommandListAppendLaunchKernel*
  - *zeCommandListAppendLaunchCooperativeKernel*
  - *zeCommandListAppendLaunchKernelIndirect*
  - *zeCommandListAppendLaunchMultipleKernelsIndirect*
- Enumerations
  - *ze_module_desc_version_t*
  - *ze_module_format_t*
  - *ze_kernel_desc_version_t*
  - *ze_kernel_flag_t*
  - *ze_kernel_attribute_t*
  - *ze_kernel_properties_version_t*
- Structures
  - *ze_module_constants_t*
  - *ze_module_desc_t*
  - *ze_kernel_desc_t*
  - *ze_kernel_properties_t*
  - *ze_group_count_t*

19.12.1 Module Functions

zeModuleCreate

__ze_api_export *ze_result_t* __zecall **zeModuleCreate**(ze_device_handle_t *hDevice*, **const** *ze_module_desc_t* **desc*, ze_module_handle_t **phModule*, ze_module_build_log_handle_t **phBuildLog*)

Creates module object from an input IL or native binary.

Parameters

- `hDevice`: handle of the device
- `desc`: pointer to module descriptor
- `phModule`: pointer to handle of module object created
- `phBuildLog`: [optional] pointer to handle of module's build log.

- Compiles the module for execution on the device.

- The module can only be used on the device on which it was created.
- The module can be copied to other devices within the same driver instance by using *zeModuleGetNativeBinary*.
- The following build options are supported:
  - ”-ze-opt-disable” - Disable optimizations
  - ”-ze-opt-greater-than-4GB-buffer-required” - Use 64-bit offset calculations for buffers.
  - ”-ze-opt-large-register-file” - Increase number of registers available to threads.
- A build log can optionally be returned to the caller. The caller is responsible for destroying build log using *zeModuleBuildLogDestroy*.
- The module descriptor constants are only supported for SPIR-V specialization constants.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == desc->pInputModule`
  - `nullptr == desc->pBuildFlags`
  - `nullptr == desc->pConstants`
  - `nullptr == phModule`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *`ZE_MODULE_DESC_VERSION_CURRENT`* `< desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->format
- *ZE_RESULT_ERROR_INVALID_NATIVE_BINARY*
- *ZE_RESULT_ERROR_INVALID_SIZE*
  - `0 == desc->inputSize`
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*
- *ZE_RESULT_ERROR_MODULE_BUILD_FAILURE*

zeModuleDestroy

__ze_api_export *ze_result_t* __zecall **`zeModuleDestroy`**(ze_module_handle_t *hModule*)

Destroys module.

Parameters

- `hModule`: [release] handle of the module

- The application is responsible for making sure the device is not currently referencing the module before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this module
- The application may **not** call this function from simultaneous threads with the same module handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeModuleBuildLogDestroy

__ze_api_export *ze_result_t* __zecall **`zeModuleBuildLogDestroy`**(ze_module_build_log_handle_t *hModuleBuildLog*)

Destroys module build log object.

Parameters

- `hModuleBuildLog`: [release] handle of the module build log object.

- The implementation of this function will immediately free all Host allocations associated with this object
- The application may **not** call this function from simultaneous threads with the same build log handle.
- The implementation of this function should be lock-free.
- This function can be called before or after *zeModuleDestroy* for the associated module.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModuleBuildLog`

- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeModuleBuildLogGetString

__ze_api_export *ze_result_t* __zecall **zeModuleBuildLogGetString**(ze_module_build_log_handle_t *hModuleBuildLog*, size_t **pSize*, char **pBuildLog*)

Retrieves text string for build log.

Parameters

- `hModuleBuildLog`: handle of the module build log object.
- `pSize`: size of build log string.
- `pBuildLog`: [optional] pointer to null-terminated string of the log.

- The caller can pass nullptr for pBuildLog when querying only for size.
- The caller must provide memory for build log.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModuleBuildLog`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pSize`

zeModuleGetNativeBinary

__ze_api_export *ze_result_t* __zecall **zeModuleGetNativeBinary**(ze_module_handle_t *hModule*, size_t **pSize*, uint8_t **pModuleNativeBinary*)

Retrieve native binary from Module.

Parameters

- `hModule`: handle of the module
- `pSize`: size of native binary in bytes.
- `pModuleNativeBinary`: [optional] byte pointer to native binary

- The native binary output can be cached to disk and new modules can be later constructed from the cached copy.
- The native binary will retain debugging information that is associated with a module.
- The caller can pass nullptr for pModuleNativeBinary when querying only for size.

- The implementation will copy the native binary into a buffer supplied by the caller.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pSize`

zeModuleGetGlobalPointer

__ze_api_export *ze_result_t* __zecall **`zeModuleGetGlobalPointer`** (ze_module_handle_t *hModule*, **`const`** char **pGlobalName*, void ***pptr*)

Retrieve global variable pointer from Module.

Parameters

- `hModule`: handle of the module
- `pGlobalName`: name of global variable in module
- `pptr`: device visible pointer

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pGlobalName`
  - `nullptr == pptr`
- *ZE_RESULT_ERROR_INVALID_GLOBAL_NAME*

zeModuleGetKernelNames

__ze_api_export *ze_result_t* __zecall **zeModuleGetKernelNames**(ze_module_handle_t *hModule*, uint32_t **pCount*, **const** char ***pNames*)

Retrieve all kernel names in the module.

Parameters

- `hModule`: handle of the module
- `pCount`: pointer to the number of names. if count is zero, then the driver will update the value with the total number of names available. if count is non-zero, then driver will only retrieve that number of names. if count is larger than the number of names available, then the driver will update the value with the correct number of names available.
- `pNames`: [optional][range(0, *pCount)] array of names of functions

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zeKernelCreate

__ze_api_export *ze_result_t* __zecall **zeKernelCreate**(ze_module_handle_t *hModule*, **const** *ze_kernel_desc_t* **desc*, ze_kernel_handle_t **phKernel*)

Create a kernel object from a module by name.

Parameters

- `hModule`: handle of the module
- `desc`: pointer to kernel descriptor
- `phKernel`: handle of the Function object

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == desc->pKernelName`
  - `nullptr == phKernel`
  - `nullptr == desc->pKernelName`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_KERNEL_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->flags
- *ZE_RESULT_ERROR_INVALID_KERNEL_NAME*

zeKernelDestroy

__ze_api_export *ze_result_t* __zecall **`zeKernelDestroy`**(ze_kernel_handle_t *hKernel*)

Destroys a kernel object.

Parameters

- `hKernel`: [release] handle of the kernel object

- All kernels must be destroyed before the module is destroyed.
- The application is responsible for making sure the device is not currently referencing the kernel before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this kernel
- The application may **not** call this function from simultaneous threads with the same kernel handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zeModuleGetFunctionPointer

__ze_api_export *ze_result_t* __zecall **zeModuleGetFunctionPointer**(ze_module_handle_t *hModule*, **const** char **pFunctionName*, void ***pfnFunction*)

Retrieve a function pointer from a module by name.

Parameters

- `hModule`: handle of the module
- `pFunctionName`: Name of function to retrieve function pointer for.
- `pfnFunction`: pointer to function.

- The function pointer is unique for the device on which the module was created.
- The function pointer is no longer valid if module is destroyed.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pFunctionName`
  - `nullptr == pfnFunction`
- *ZE_RESULT_ERROR_INVALID_FUNCTION_NAME*

zeKernelSetGroupSize

__ze_api_export *ze_result_t* __zecall **zeKernelSetGroupSize**(ze_kernel_handle_t *hKernel*, uint32_t *groupSizeX*, uint32_t *groupSizeY*, uint32_t *groupSizeZ*)

Set group size for a kernel.

Parameters

- `hKernel`: handle of the kernel object
- `groupSizeX`: group size for X dimension to use for this kernel
- `groupSizeY`: group size for Y dimension to use for this kernel
- `groupSizeZ`: group size for Z dimension to use for this kernel

- The application may **not** call this function from simultaneous threads with the same kernel handle.
- The implementation of this function should be lock-free.

- The implementation will copy the group size information into a command list when the function is appended.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_GROUP_SIZE_DIMENSION*

zeKernelSuggestGroupSize

__ze_api_export *ze_result_t* __zecall **`zeKernelSuggestGroupSize`**(ze_kernel_handle_t *hKernel*, uint32_t *globalSizeX*, uint32_t *globalSizeY*, uint32_t *globalSizeZ*, uint32_t **groupSizeX*, uint32_t **groupSizeY*, uint32_t **groupSizeZ*)

Query a suggested group size for a kernel given a global size for each dimension.

Parameters

- `hKernel`: handle of the kernel object
- `globalSizeX`: global width for X dimension
- `globalSizeY`: global width for Y dimension
- `globalSizeZ`: global width for Z dimension
- `groupSizeX`: recommended size of group for X dimension
- `groupSizeY`: recommended size of group for Y dimension
- `groupSizeZ`: recommended size of group for Z dimension

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.
- This function ignores the group size that is set using *zeKernelSetGroupSize*.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == groupSizeX`

    - `nullptr == groupSizeY`
    - `nullptr == groupSizeZ`
  - *ZE_RESULT_ERROR_INVALID_GLOBAL_WIDTH_DIMENSION*

zeKernelSuggestMaxCooperativeGroupCount

__ze_api_export *ze_result_t* __zecall **zeKernelSuggestMaxCooperativeGroupCount**(ze_kernel_handle_t *hKernel*, uint32_t **totalGroupCount*)

Query a suggested max group count a cooperative kernel.

Parameters

  - `hKernel`: handle of the kernel object
  - `totalGroupCount`: recommended total group count.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

  - *ZE_RESULT_SUCCESS*
  - *ZE_RESULT_ERROR_UNINITIALIZED*
  - *ZE_RESULT_ERROR_DEVICE_LOST*
  - *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hKernel`
  - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == totalGroupCount`

zeKernelSetArgumentValue

__ze_api_export *ze_result_t* __zecall **zeKernelSetArgumentValue**(ze_kernel_handle_t *hKernel*, uint32_t *argIndex*, size_t *argSize*, **const** void **pArgValue*)

Set kernel argument used on kernel launch.

Parameters

  - `hKernel`: handle of the kernel object
  - `argIndex`: argument index in range [0, num args - 1]
  - `argSize`: size of argument type
  - `pArgValue`: [optional] argument value represented as matching arg type. If null then argument value is considered null.

- This function may **not** be called from simultaneous threads with the same function handle.

- The implementation of this function should be lock-free.
- The implementation will copy the arguments into a command list when the function is appended.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_INDEX*
- *ZE_RESULT_ERROR_INVALID_KERNEL_ARGUMENT_SIZE*

zeKernelSetAttribute

__ze_api_export *ze_result_t* __zecall **zeKernelSetAttribute**(ze_kernel_handle_t *hKernel*, *ze_kernel_attribute_t* *attr*, uint32_t *size*, **const** void **pValue*)

Sets a kernel attribute.

Parameters

- `hKernel`: handle of the kernel object
- `attr`: attribute to set
- `size`: size in bytes of kernel attribute value.
- `pValue`: [optional] pointer to attribute value.

- This function may **not** be called from simultaneous threads with the same function handle.
- The implementation of this function should be lock-free.

Remark *Analogues*

- **clSetKernelExecInfo**

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - attr
- *ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE*

zeKernelGetAttribute

__ze_api_export *ze_result_t* __zecall **zeKernelGetAttribute**(ze_kernel_handle_t *hKernel*, *ze_kernel_attribute_t* *attr*, uint32_t **pSize*, void **pValue*)

Gets a kernel attribute.

Parameters

- `hKernel`: handle of the kernel object
- `attr`: attribute to get. Documentation for *ze_kernel_attribute_t* for return type information for pValue.
- `pSize`: size in bytes needed for kernel attribute value. If pValue is nullptr then the size needed for pValue memory will be written to pSize. Only need to query size for arbitrary sized attributes.
- `pValue`: [optional] pointer to attribute value result.

- This function may **not** be called from simultaneous threads with the same function handle.
- The implementation of this function should be lock-free.
- The caller sets pValue to nullptr when querying only for size.
- The caller must provide memory for pValue querying when querying size.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - attr
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pSize`
- *ZE_RESULT_ERROR_INVALID_KERNEL_ATTRIBUTE_VALUE*

zeKernelSetIntermediateCacheConfig

__ze_api_export *ze_result_t* __zecall **zeKernelSetIntermediateCacheConfig**(ze_kernel_handle_t *hKernel*, *ze_cache_config_t* *CacheConfig*)

Sets the preferred Intermediate cache configuration for a kernel.

Parameters

- `hKernel`: handle of the kernel object
- `CacheConfig`: CacheConfig

- The application may **not** call this function from simultaneous threads with the same kernel handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - CacheConfig
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*

zeKernelGetProperties

__ze_api_export *ze_result_t* __zecall **zeKernelGetProperties** (ze_kernel_handle_t *hKernel*, *ze_kernel_properties_t* **pKernelProperties*)

Retrieve kernel properties.

Parameters

- `hKernel`: handle of the kernel object
- `pKernelProperties`: query result for kernel properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pKernelProperties`

zeCommandListAppendLaunchKernel

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendLaunchKernel**(ze_command_list_handle_t *hCommandList*, ze_kernel_handle_t *hKernel*, **const** *ze_group_count_t* **pLaunchFuncArgs*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Launch kernel over one or more work groups.

Parameters

- `hCommandList`: handle of the command list
- `hKernel`: handle of the kernel object
- `pLaunchFuncArgs`: thread group launch arguments
- `hSignalEvent`: [optional] handle of the event to signal on completion
- `numWaitEvents`: [optional] number of events to wait on before launching
- `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before launching

- This may **only** be called for a command list created with command queue group ordinal that supports compute.
- This function may **not** be called from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pLaunchFuncArgs`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendLaunchCooperativeKernel

__ze_api_export *ze_result_t* __zecall **zeCommandListAppendLaunchCooperativeKernel**(ze_command_list_handle_t *hCommandList*, ze_kernel_handle_t *hKernel*, **const** *ze_group_count_t* **pLaunchFuncArgs*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Launch kernel cooperatively over one or more work groups.

Parameters

- `hCommandList`: handle of the command list
- `hKernel`: handle of the kernel object
- `pLaunchFuncArgs`: thread group launch arguments
- `hSignalEvent`: [optional] handle of the event to signal on completion
- `numWaitEvents`: [optional] number of events to wait on before launching
- `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before launching

- This may **only** be called for a command list created with command queue group ordinal that supports compute.
- This may only be used for a command list that are submitted to command queue with cooperative flag set.
- This function may **not** be called from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.
- Use *zeKernelSuggestMaxCooperativeGroupCount* to recommend max group count for device for cooperative functions that device supports.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

  - `nullptr == hCommandList`
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pLaunchFuncArgs`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zeCommandListAppendLaunchKernelIndirect

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendLaunchKernelIndirect`** (ze_command_list_handle_t *hCommandList*, ze_kernel_handle_t *hKernel*, **`const`** *ze_group_count_t* **pLaunchArgumentsBuffer*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Launch kernel over one or more work groups using indirect arguments.

Parameters

- `hCommandList`: handle of the command list
- `hKernel`: handle of the kernel object
- `pLaunchArgumentsBuffer`: pointer to device buffer that will contain thread group launch arguments
- `hSignalEvent`: [optional] handle of the event to signal on completion
- `numWaitEvents`: [optional] number of events to wait on before launching
- `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before launching

- The launch arguments need to be device visible.
- The launch arguments buffer may not be reused until the function has completed on the device.
- This may **only** be called for a command list created with command queue group ordinal that supports compute.
- This function may **not** be called from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pLaunchArgumentsBuffer`
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT`*

zeCommandListAppendLaunchMultipleKernelsIndirect

__ze_api_export *ze_result_t* __zecall **`zeCommandListAppendLaunchMultipleKernelsIndirect`** (ze_command_list_handl *hCommandList*, uint32_t *numKernels*, ze_kernel_handle_t **phKernels*, **const** uint32_t **pCountBuffer*, **const** *ze_group_count_t* **pLaunchArgumentsBuffer*, ze_event_handle_t *hSignalEvent*, uint32_t *numWaitEvents*, ze_event_handle_t **phWaitEvents*)

Launch multiple kernels over one or more work groups using an array of indirect arguments.

Parameters

   - `hCommandList`: handle of the command list
   - `numKernels`: maximum number of kernels to launch
   - `phKernels`: [range(0, numKernels)] handles of the kernel objects
   - `pCountBuffer`: pointer to device memory location that will contain the actual number of kernels to launch; value must be less-than or equal-to numKernels
   - `pLaunchArgumentsBuffer`: [range(0, numKernels)] pointer to device buffer that will contain a contiguous array of thread group launch arguments
   - `hSignalEvent`: [optional] handle of the event to signal on completion
   - `numWaitEvents`: [optional] number of events to wait on before launching
   - `phWaitEvents`: [optional][range(0, numWaitEvents)] handle of the events to wait on before launching

- The array of launch arguments need to be device visible.
- The array of launch arguments buffer may not be reused until the kernel has completed on the device.
- This may **only** be called for a command list created with command queue group ordinal that supports compute.
- This function may **not** be called from simultaneous threads with the same command list handle.
- The implementation of this function should be lock-free.

Return

   - *ZE_RESULT_SUCCESS*
   - *ZE_RESULT_ERROR_UNINITIALIZED*
   - *ZE_RESULT_ERROR_DEVICE_LOST*
   - *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
     - `nullptr == hCommandList`
   - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
     - `nullptr == phKernels`
     - `nullptr == pCountBuffer`
     - `nullptr == pLaunchArgumentsBuffer`
   - *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

19.12.2 Module Enums

ze_module_desc_version_t

enum ze_module_desc_version_t
API version of *ze_module_desc_t*.

Values:

enumerator ZE_MODULE_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_module_format_t

enum ze_module_format_t

Supported module creation input formats.

Values:

enumerator ZE_MODULE_FORMAT_IL_SPIRV = 0

Format is SPIRV IL format.

enumerator ZE_MODULE_FORMAT_NATIVE

Format is device native format.

ze_kernel_desc_version_t

enum ze_kernel_desc_version_t

API version of *ze_kernel_desc_t*.

Values:

enumerator ZE_KERNEL_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

ze_kernel_flag_t

enum ze_kernel_flag_t

Supported kernel creation flags.

Values:

enumerator ZE_KERNEL_FLAG_NONE = 0

default driver behavior

enumerator ZE_KERNEL_FLAG_FORCE_RESIDENCY

force all device allocations to be resident during execution

ze_kernel_attribute_t

enum ze_kernel_attribute_t

Kernel attributes.

Remark *Analogues*

- **cl_kernel_exec_info**

Values:

enumerator ZE_KERNEL_ATTR_INDIRECT_HOST_ACCESS = 0

Indicates that the function accesses host allocations indirectly (default: false, type: bool_t)

enumerator ZE_KERNEL_ATTR_INDIRECT_DEVICE_ACCESS

Indicates that the function accesses device allocations indirectly (default: false, type: bool_t)

enumerator ZE_KERNEL_ATTR_INDIRECT_SHARED_ACCESS

Indicates that the function accesses shared allocations indirectly (default: false, type: bool_t)

enumerator ZE_KERNEL_ATTR_SOURCE_ATTRIBUTE
Declared kernel attributes (i.e. can be specified with **attribute** in runtime language). (type: char[]) Returned as a null-terminated string and each attribute is separated by a space. *zeKernelSetAttribute* is not supported for this.

ze_kernel_properties_version_t

enum ze_kernel_properties_version_t
API version of *ze_kernel_properties_t*.

Values:

enumerator ZE_KERNEL_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

19.12.3 Module Structures

ze_module_constants_t

struct ze_module_constants_t
Specialization constants - User defined constants.

Public Members

uint32_t **numConstants**
[in] Number of specialization constants.

const uint32_t ***pConstantIds**
[in] Pointer to array of IDs that is sized to numConstants.

const uint64_t ***pConstantValues**
[in] Pointer to array of values that is sized to numConstants.

ze_module_desc_t

struct ze_module_desc_t
Module descriptor.

Public Members

ze_module_desc_version_t **version**
[in] *ZE_MODULE_DESC_VERSION_CURRENT*

ze_module_format_t **format**
[in] Module format passed in with pInputModule

size_t **inputSize**
[in] size of input IL or ISA from pInputModule.

const uint8_t ***pInputModule**
[in] pointer to IL or ISA

const char ***pBuildFlags**
[in] string containing compiler flags. See programming guide for build flags.

const *ze_module_constants_t* ***pConstants**
: [in] pointer to specialization constants. Valid only for SPIR-V input. This must be set to nullptr if no specialization constants are provided.

ze_kernel_desc_t

struct ze_kernel_desc_t
Kernel descriptor.

Public Members

ze_kernel_desc_version_t **version**
: [in] *ZE_KERNEL_DESC_VERSION_CURRENT*

ze_kernel_flag_t **flags**
: [in] creation flags

const char ***pKernelName**
: [in] null-terminated name of kernel in module

ze_kernel_properties_t

struct ze_kernel_properties_t
Kernel properties.

Public Members

ze_kernel_properties_version_t **version**
: [in] *ZE_KERNEL_PROPERTIES_VERSION_CURRENT*

char **name**[**ZE_MAX_KERNEL_NAME**]
: [out] Kernel name

uint32_t **numKernelArgs**
: [out] number of kernel arguments.

uint32_t **requiredGroupSizeX**
: [out] required group size in the X dimension

uint32_t **requiredGroupSizeY**
: [out] required group size in the Y dimension

uint32_t **requiredGroupSizeZ**
: [out] required group size in the Z dimension

ze_group_count_t

struct ze_group_count_t

Kernel dispatch group count.

Public Members

uint32_t **groupCountX**

[in] number of thread groups in X dimension

uint32_t **groupCountY**

[in] number of thread groups in Y dimension

uint32_t **groupCountZ**

[in] number of thread groups in Z dimension

19.13 Residency

- Functions
  - *zeDeviceMakeMemoryResident*
  - *zeDeviceEvictMemory*
  - *zeDeviceMakeImageResident*
  - *zeDeviceEvictImage*

19.13.1 Residency Functions

zeDeviceMakeMemoryResident

__ze_api_export *ze_result_t* __zecall **zeDeviceMakeMemoryResident**(ze_device_handle_t *hDevice*, void **ptr*, size_t *size*)

Makes memory resident for the device.

Parameters

- `hDevice`: handle of the device
- `ptr`: pointer to memory to make resident
- `size`: size in bytes to make resident

- If the application does not properly manage residency then the device may experience unrecoverable page-faults.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*

- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDeviceEvictMemory

__ze_api_export *ze_result_t* __zecall **zeDeviceEvictMemory**(ze_device_handle_t *hDevice*, void **ptr*, size_t *size*)

Allows memory to be evicted from the device.

Parameters

- `hDevice`: handle of the device
- `ptr`: pointer to memory to evict
- `size`: size in bytes to evict

- The application is responsible for making sure the device is not currently referencing the memory before it is evicted
- Memory is always implicitly evicted if it is resident when freed.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == ptr`
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDeviceMakeImageResident

__ze_api_export *ze_result_t* __zecall **`zeDeviceMakeImageResident`** (ze_device_handle_t *hDevice*, ze_image_handle_t *hImage*)

Makes image resident for the device.

Parameters

- `hDevice`: handle of the device
- `hImage`: handle of image to make resident

- If the application does not properly manage residency then the device may experience unrecoverable page-faults.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
  - `nullptr == hImage`
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

zeDeviceEvictImage

__ze_api_export *ze_result_t* __zecall **`zeDeviceEvictImage`** (ze_device_handle_t *hDevice*, ze_image_handle_t *hImage*)

Allows image to be evicted from the device.

Parameters

- `hDevice`: handle of the device
- `hImage`: handle of image to make evict

- The application is responsible for making sure the device is not currently referencing the memory before it is evicted
- An image is always implicitly evicted if it is resident when destroyed.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
  - `nullptr == hImage`
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
- *ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY*

19.14 Sampler

- Functions
  - *zeSamplerCreate*
  - *zeSamplerDestroy*
- Enumerations
  - *ze_sampler_desc_version_t*
  - *ze_sampler_address_mode_t*
  - *ze_sampler_filter_mode_t*
- Structures
  - *ze_sampler_desc_t*

19.14.1 Sampler Functions

zeSamplerCreate

__ze_api_export *ze_result_t* __zecall **`zeSamplerCreate`**(ze_device_handle_t *hDevice*, **`const`** *ze_sampler_desc_t* **desc*, ze_sampler_handle_t **phSampler*)

Creates sampler object.

Parameters

- `hDevice`: handle of the device
- `desc`: pointer to sampler descriptor
- `phSampler`: handle of the sampler

- The sampler can only be used on the device on which it was created.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*

- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phSampler`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZE_SAMPLER_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - desc->addressMode
  - desc->filterMode
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*

zeSamplerDestroy

__ze_api_export *ze_result_t* __zecall **`zeSamplerDestroy`**(ze_sampler_handle_t *hSampler*)

Destroys sampler object.

Parameters

- `hSampler`: [release] handle of the sampler

- The application is responsible for making sure the device is not currently referencing the sampler before it is deleted
- The implementation of this function will immediately free all Host and Device allocations associated with this module
- The application may **not** call this function from simultaneous threads with the same sampler handle.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSampler`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

19.14.2 Sampler Enums

ze_sampler_desc_version_t

enum ze_sampler_desc_version_t

API version of *ze_sampler_desc_t*.

Values:

enumerator ZE_SAMPLER_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

ze_sampler_address_mode_t

enum ze_sampler_address_mode_t

Sampler addressing modes.

Values:

enumerator ZE_SAMPLER_ADDRESS_MODE_NONE = 0
No coordinate modifications for out-of-bounds image access.

enumerator ZE_SAMPLER_ADDRESS_MODE_REPEAT
Out-of-bounds coordinates are wrapped back around.

enumerator ZE_SAMPLER_ADDRESS_MODE_CLAMP
Out-of-bounds coordinates are clamped to edge.

enumerator ZE_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
Out-of-bounds coordinates are clamped to border color which is (0.0f, 0.0f, 0.0f, 0.0f) if image format swizzle contains alpha, otherwise (0.0f, 0.0f, 0.0f, 1.0f).

enumerator ZE_SAMPLER_ADDRESS_MODE_MIRROR
Out-of-bounds coordinates are mirrored starting from edge.

ze_sampler_filter_mode_t

enum ze_sampler_filter_mode_t

Sampler filtering modes.

Values:

enumerator ZE_SAMPLER_FILTER_MODE_NEAREST = 0
No coordinate modifications for out of bounds image access.

enumerator ZE_SAMPLER_FILTER_MODE_LINEAR
Out-of-bounds coordinates are wrapped back around.

19.14.3 Sampler Structures

ze_sampler_desc_t

`struct ze_sampler_desc_t`

Sampler descriptor.

Public Members

ze_sampler_desc_version_t **`version`**

[in] *ZE_SAMPLER_DESC_VERSION_CURRENT*

ze_sampler_address_mode_t **`addressMode`**

[in] Sampler addressing mode to determine how out-of-bounds coordinates are handled.

ze_sampler_filter_mode_t **`filterMode`**

[in] Sampler filter mode to determine how samples are filtered.

ze_bool_t **`isNormalized`**

[in] Are coordinates normalized [0, 1] or not.

CHAPTER

TWENTY

TOOLS API

oneAPI Level Zero Specification - Version 0.91

20.1 Driver

- Functions
  - *zetInit*

20.1.1 Driver Functions

zetInit

__ze_api_export *ze_result_t* __zecall **`zetInit`** (*ze_init_flag_t flags*)

Initialize the 'One API' driver and must be called before any other API function.

Parameters

- `flags`: initialization flags

- If this function is not called then all other functions will return *ZE_RESULT_ERROR_UNINITIALIZED*.
- Only one instance of a driver per process will be initialized.
- This function is thread-safe for scenarios where multiple libraries may initialize the driver simultaneously.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - flags
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*

20.2 Debug

- Functions
  - *zetDebugAttach*
  - *zetDebugDetach*
  - *zetDebugGetNumThreads*
  - *zetDebugReadEvent*
  - *zetDebugInterrupt*
  - *zetDebugResume*
  - *zetDebugReadMemory*
  - *zetDebugWriteMemory*
  - *zetDebugReadState*
  - *zetDebugWriteState*
- Enumerations
  - *zet_debug_event_flags_t*
  - *zet_debug_event_type_t*
  - *zet_debug_detach_reason_t*
  - *zet_debug_memory_space_intel_graphics_t*
  - *zet_debug_state_intel_graphics_t*
- Structures
  - *zet_debug_config_v1_t*
  - *zet_debug_config_variants_t*
  - *zet_debug_config_t*
  - *zet_debug_event_info_detached_t*
  - *zet_debug_event_info_module_t*
  - *zet_debug_event_info_t*
  - *zet_debug_event_t*
  - *zet_debug_state_section_t*
  - *zet_debug_state_t*

20.2.1 Debug Functions

zetDebugAttach

__ze_api_export *ze_result_t* __zecall **zetDebugAttach**(zet_device_handle_t *hDevice*, **const** *zet_debug_config_t* **config*, zet_debug_session_handle_t **hDebug*)

Attach to a device.

Parameters

- `hDevice`: device handle
- `config`: the debug configuration
- `hDebug`: debug session handle

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == config`
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid device handle has been supplied
  - an invalid configuration has been supplied
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - attaching to this device is not supported
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - caller does not have sufficient permissions
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - a debugger is already attached

zetDebugDetach

__ze_api_export *ze_result_t* __zecall **zetDebugDetach**(zet_debug_session_handle_t *hDebug*)

Close a debug session.

Parameters

- `hDebug`: [release] debug session handle

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle has been supplied

zetDebugGetNumThreads

__ze_api_export *ze_result_t* __zecall **zetDebugGetNumThreads** (zet_debug_session_handle_t *hDebug*, uint64_t **pNumThreads*)

Query the number of device threads for a debug session.

Parameters

- `hDebug`: debug session handle
- `pNumThreads`: the maximal number of threads

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pNumThreads`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle has been supplied

zetDebugReadEvent

__ze_api_export *ze_result_t* __zecall **zetDebugReadEvent** (zet_debug_session_handle_t *hDebug*, uint64_t *timeout*, size_t *size*, void **buffer*)

Read the topmost debug event.

Parameters

- `hDebug`: debug session handle
- `timeout`: timeout in milliseconds (or ZET_DEBUG_TIMEOUT_INFINITE)
- `size`: the size of the buffer in bytes
- `buffer`: a buffer to hold the event data

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == buffer`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or buffer pointer has been supplied
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*
  - the output buffer is too small to hold the event
- *ZE_RESULT_NOT_READY*
  - the timeout expired

zetDebugInterrupt

__ze_api_export *ze_result_t* __zecall **`zetDebugInterrupt`** (zet_debug_session_handle_t *hDebug*, uint64_t *threadid*)

Interrupt device threads.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread to inerrupt or ZET_DEBUG_THREAD_ALL

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is already stopped or unavailable

zetDebugResume

__ze_api_export *ze_result_t* __zecall **zetDebugResume** (zet_debug_session_handle_t *hDebug*, uint64_t *threadid*)

Resume device threads.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread to resume or ZET_DEBUG_THREAD_ALL

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is already running or unavailable

zetDebugReadMemory

__ze_api_export *ze_result_t* __zecall **zetDebugReadMemory** (zet_debug_session_handle_t *hDebug*, uint64_t *threadid*, int *memSpace*, uint64_t *address*, size_t *size*, void **buffer*)

Read memory.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread context or ZET_DEBUG_THREAD_NONE
- `memSpace`: the (device-specific) memory space
- `address`: the virtual address of the memory to read from
- `size`: the number of bytes to read
- `buffer`: a buffer to hold a copy of the memory

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

  - `nullptr == buffer`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is running or unavailable
  - an invalid address has been supplied
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - the memory cannot be accessed from the supplied thread

zetDebugWriteMemory

__ze_api_export *ze_result_t* __zecall **zetDebugWriteMemory** (zet_debug_session_handle_t *hDebug*, uint64_t *threadid*, int *memSpace*, uint64_t *address*, size_t *size*, **const** void **buffer*)

Write memory.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread context or ZET_DEBUG_THREAD_NONE
- `memSpace`: the (device-specific) memory space
- `address`: the virtual address of the memory to write to
- `size`: the number of bytes to write
- `buffer`: a buffer holding the pattern to write

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == buffer`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is running or unavailable
  - an invalid address has been supplied
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - the memory cannot be accessed from the supplied thread

zetDebugReadState

__ze_api_export *ze_result_t* __zecall **zetDebugReadState**(zet_debug_session_handle_t *hDebug*, uint64_t *threadid*, uint64_t *offset*, size_t *size*, void **buffer*)

Read register state.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread context
- `offset`: the offset into the register state area
- `size`: the number of bytes to read
- `buffer`: a buffer to hold a copy of the register state

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == buffer`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is running or unavailable
  - an invalid offset or size has been supplied

zetDebugWriteState

__ze_api_export *ze_result_t* __zecall **zetDebugWriteState**(zet_debug_session_handle_t *hDebug*, uint64_t *threadid*, uint64_t *offset*, size_t *size*, **const** void **buffer*)

Write register state.

Parameters

- `hDebug`: debug session handle
- `threadid`: the thread context
- `offset`: the offset into the register state area
- `size`: the number of bytes to write
- `buffer`: a buffer holding the pattern to write

Return

- *ZE_RESULT_SUCCESS*

- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDebug`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == buffer`
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - an invalid debug handle or thread identifier has been supplied
  - the thread is running or unavailable
  - an invalid offset or size has been supplied

20.2.2 Debug Enums

zet_debug_event_flags_t

enum zet_debug_event_flags_t
Debug event flags.

Values:

enumerator ZET_DEBUG_EVENT_FLAG_NONE = 0
No event flags.

enumerator ZET_DEBUG_EVENT_FLAG_STOPPED = ZE_BIT(0)
The reporting thread stopped.

zet_debug_event_type_t

enum zet_debug_event_type_t
Debug event types.

Values:

enumerator ZET_DEBUG_EVENT_INVALID = 0
The event is invalid.

enumerator ZET_DEBUG_EVENT_DETACHED
The tool was detached.

enumerator ZET_DEBUG_EVENT_PROCESS_ENTRY
The debuggee process created command queues on the device.

enumerator ZET_DEBUG_EVENT_PROCESS_EXIT
The debuggee process destroyed all command queues on the device.

enumerator ZET_DEBUG_EVENT_MODULE_LOAD
An in-memory module was loaded onto the device.

enumerator ZET_DEBUG_EVENT_MODULE_UNLOAD
An in-memory module is about to get unloaded from the device.

enumerator ZET_DEBUG_EVENT_EXCEPTION
The thread stopped due to a device exception.

zet_debug_detach_reason_t

`enum zet_debug_detach_reason_t`

Debug detach reason.

Values:

`enumerator ZET_DEBUG_DETACH_INVALID` = 0

The detach reason is not valid.

`enumerator ZET_DEBUG_DETACH_HOST_EXIT`

The host process exited.

zet_debug_memory_space_intel_graphics_t

`enum zet_debug_memory_space_intel_graphics_t`

Memory spaces for Intel Graphics devices.

Values:

`enumerator ZET_DEBUG_MEMORY_SPACE_GEN_DEFAULT` = 0

default memory space (attribute may be omitted)

`enumerator ZET_DEBUG_MEMORY_SPACE_GEN_SLM`

shared local memory space

zet_debug_state_intel_graphics_t

`enum zet_debug_state_intel_graphics_t`

Register file types for Intel Graphics devices.

Values:

`enumerator ZET_DEBUG_STATE_GEN_INVALID` = 0

An invalid register file.

`enumerator ZET_DEBUG_STATE_GEN_GRF`

The general register file.

`enumerator ZET_DEBUG_STATE_GEN_ACC`

The accumulator register file.

`enumerator ZET_DEBUG_STATE_GEN_ADDR`

The address register file.

`enumerator ZET_DEBUG_STATE_GEN_FLAG`

The flags register file.

20.2.3 Debug Structures

zet_debug_config_v1_t

struct zet_debug_config_v1_t
Debug configuration: version 1.

Public Members

int **pid**
The host process identifier.

zet_debug_config_variants_t

union zet_debug_config_variants_t
#include <zet_debug.h> Debug configuration: version-dependent fields.

Public Members

zet_debug_config_v1_t **v1**
Version 1.

zet_debug_config_t

struct zet_debug_config_t
Debug configuration.

Public Members

uint16_t **version**
The requested program debug API version.

zet_debug_config_variants_t **variant**
Version-specific fields.

zet_debug_event_info_detached_t

struct zet_debug_event_info_detached_t
Event information for *ZET_DEBUG_EVENT_DETACHED*.

Public Members

uint8_t **reason**
The detach reason.

zet_debug_event_info_module_t

struct zet_debug_event_info_module_t
Event information for *ZET_DEBUG_EVENT_MODULE_LOAD*/UNLOAD.

Public Members

uint64_t **moduleBegin**
The begin address of the in-memory module.

uint64_t **moduleEnd**
The end address of the in-memory module.

uint64_t **load**
The load address of the module on the device.

zet_debug_event_info_t

union zet_debug_event_info_t
#include <zet_debug.h> Event type specific information.

Public Members

zet_debug_event_info_detached_t **detached**
type == *ZET_DEBUG_EVENT_DETACHED*

zet_debug_event_info_module_t **module**
type == *ZET_DEBUG_EVENT_MODULE_LOAD*/UNLOAD

zet_debug_event_t

struct zet_debug_event_t
A debug event on the device.

Public Members

uint8_t **type**
The event type.

uint64_t **thread**
The thread reporting the event.

uint64_t **flags**
A bit-vector of *zet_debug_event_flags_t*.

zet_debug_event_info_t **info**
Event type specific information.

zet_debug_state_section_t

struct zet_debug_state_section_t
A register file descriptor.

Public Members

uint16_t **type**
The register file type type.

uint16_t **version**
The register file version.

uint32_t **size**
The size of the register file in bytes.

uint64_t **offset**
The offset into the register state area.

zet_debug_state_t

struct zet_debug_state_t
A register state descriptor.

Public Members

uint32_t **size**
The size of the register state object in bytes.

uint8_t **headerSize**
The size of the register state descriptor in bytes.

uint8_t **secSize**
The size of the register file descriptors in bytes.

uint16_t **numSec**
The number of register file descriptors.

20.3 Metric

- Functions
  - *zetMetricGroupGet*
  - *zetMetricGroupGetProperties*
  - *zetMetricGroupCalculateMetricValues*
  - *zetMetricGet*
  - *zetMetricGetProperties*
  - *zetDeviceActivateMetricGroups*
  - *zetMetricTracerOpen*
  - *zetCommandListAppendMetricTracerMarker*

  - *zetMetricTracerClose*
  - *zetMetricTracerReadData*
  - *zetMetricQueryPoolCreate*
  - *zetMetricQueryPoolDestroy*
  - *zetMetricQueryCreate*
  - *zetMetricQueryDestroy*
  - *zetMetricQueryReset*
  - *zetCommandListAppendMetricQueryBegin*
  - *zetCommandListAppendMetricQueryEnd*
  - *zetCommandListAppendMetricMemoryBarrier*
  - *zetMetricQueryGetData*
- Enumerations
  - *zet_metric_group_sampling_type_t*
  - *zet_metric_group_properties_version_t*
  - *zet_metric_type_t*
  - *zet_value_type_t*
  - *zet_metric_properties_version_t*
  - *zet_metric_tracer_desc_version_t*
  - *zet_metric_query_pool_desc_version_t*
  - *zet_metric_query_pool_flag_t*
- Structures
  - *zet_metric_group_properties_t*
  - *zet_value_t*
  - *zet_typed_value_t*
  - *zet_metric_properties_t*
  - *zet_metric_tracer_desc_t*
  - *zet_metric_query_pool_desc_t*

20.3.1 Metric Functions

zetMetricGroupGet

__ze_api_export *ze_result_t* __zecall **`zetMetricGroupGet`** (zet_device_handle_t *hDevice*, uint32_t **pCount*, zet_metric_group_handle_t **phMetricGroups*)

Retrieves metric group for a device.

Parameters

- `hDevice`: handle of the device

    - `pCount`: pointer to the number of metric groups. if count is zero, then the driver will update the value with the total number of metric groups available. if count is non-zero, then driver will only retrieve that number of metric groups. if count is larger than the number of metric groups available, then the driver will update the value with the correct number of metric groups available.
    - `phMetricGroups`: [optional][range(0, *pCount)] array of handle of metric groups

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pCount`

zetMetricGroupGetProperties

__ze_api_export *ze_result_t* __zecall **zetMetricGroupGetProperties**(zet_metric_group_handle_t *hMetricGroup*, *zet_metric_group_properties_t* **pProperties*)

Retrieves attributes of a metric group.

Parameters

- `hMetricGroup`: handle of the metric group
- `pProperties`: metric group properties

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hMetricGroup`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pProperties`

zetMetricGroupCalculateMetricValues

__ze_api_export *ze_result_t* __zecall **zetMetricGroupCalculateMetricValues**(zet_metric_group_handle_t *hMetricGroup*, size_t *rawDataSize*, **const** uint8_t **pRawData*, uint32_t **pMetricValueCount*, *zet_typed_value_t* **pMetricValues*)

Calculates metric values from raw data.

Parameters

- `hMetricGroup`: handle of the metric group
- `rawDataSize`: size in bytes of raw data buffer
- `pRawData`: [range(0, rawDataSize)] buffer of raw data to calculate
- `pMetricValueCount`: pointer to number of metric values calculated. if count is zero, then the driver will update the value with the total number of metric values to be calculated. if count is non-zero, then driver will only calculate that number of metric values. if count is larger than the number available in the raw data buffer, then the driver will update the value with the actual number of metric values to be calculated.
- `pMetricValues`: [optional][range(0, *pMetricValueCount)] buffer of calculated metrics

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricGroup`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pRawData`
  - `nullptr == pMetricValueCount`

zetMetricGet

__ze_api_export *ze_result_t* __zecall **zetMetricGet** (zet_metric_group_handle_t *hMetricGroup*, uint32_t **pCount*, zet_metric_handle_t **phMetrics*)

Retrieves metric from a metric group.

Parameters

- `hMetricGroup`: handle of the metric group
- `pCount`: pointer to the number of metrics. if count is zero, then the driver will update the value with the total number of metrics available. if count is non-zero, then driver will only retrieve that number of metrics. if count is larger than the number of metrics available, then the driver will update the value with the correct number of metrics available.
- `phMetrics`: [optional][range(0, *pCount)] array of handle of metrics

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricGroup`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetMetricGetProperties

__ze_api_export *ze_result_t* __zecall **zetMetricGetProperties** (zet_metric_handle_t *hMetric*, *zet_metric_properties_t* **pProperties*)

Retrieves attributes of a metric.

Parameters

- `hMetric`: handle of the metric
- `pProperties`: metric properties

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetric`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

    - `nullptr == pProperties`

zetDeviceActivateMetricGroups

__ze_api_export *ze_result_t* __zecall **zetDeviceActivateMetricGroups**(zet_device_handle_t *hDevice*, uint32_t *count*, zet_metric_group_handle_t **phMetricGroups*)

Activates metric groups.

Parameters

- `hDevice`: handle of the device
- `count`: metric group count to activate. 0 to deactivate.
- `phMetricGroups`: [optional][range(0, count)] handles of the metric groups to activate. NULL to deactivate.

- MetricGroup must be active until MetricQueryGetDeta and *zetMetricTracerClose*.
- Conflicting metric groups cannot be activated, in such case tha call would fail.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hDevice`

zetMetricTracerOpen

__ze_api_export *ze_result_t* __zecall **zetMetricTracerOpen**(zet_device_handle_t *hDevice*, zet_metric_group_handle_t *hMetricGroup*, *zet_metric_tracer_desc_t* **desc*, ze_event_handle_t *hNotificationEvent*, zet_metric_tracer_handle_t **phMetricTracer*)

Opens metric tracer for a device.

Parameters

- `hDevice`: handle of the device
- `hMetricGroup`: handle of the metric group
- `desc`: metric tracer descriptor
- `hNotificationEvent`: [optional] event used for report availability notification. Must be device to host type.
- `phMetricTracer`: handle of metric tracer

- The application may **not** call this function from simultaneous threads with the same device handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
  - `nullptr == hMetricGroup`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == phMetricTracer`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - *`ZET_METRIC_TRACER_DESC_VERSION_CURRENT`* `< desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
- *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zetCommandListAppendMetricTracerMarker

__ze_api_export *ze_result_t* __zecall **`zetCommandListAppendMetricTracerMarker`**(zet_command_list_handle_t *hCommandList*, zet_metric_tracer_handle_t *hMetricTracer*, uint32_t *value*)

Append metric tracer marker into a command list.

Parameters

- `hCommandList`: handle of the command list
- `hMetricTracer`: handle of the metric tracer
- `value`: tracer marker value

- The application may **not** call this function from simultaneous threads with the same command list handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hMetricTracer`

zetMetricTracerClose

__ze_api_export *ze_result_t* __zecall **zetMetricTracerClose**(zet_metric_tracer_handle_t *hMetricTracer*)

Closes metric tracer.

Parameters

- `hMetricTracer`: [release] handle of the metric tracer

- The application may **not** call this function from simultaneous threads with the same metric tracer handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricTracer`

zetMetricTracerReadData

__ze_api_export *ze_result_t* __zecall **zetMetricTracerReadData**(zet_metric_tracer_handle_t *hMetricTracer*, uint32_t *maxReportCount*, size_t **pRawDataSize*, uint8_t **pRawData*)

Reads data from metric tracer.

Parameters

- `hMetricTracer`: handle of the metric tracer
- `maxReportCount`: the maximum number of reports the application wants to receive. if UINT32_MAX, then function will retrieve all reports available
- `pRawDataSize`: pointer to size in bytes of raw data requested to read. if size is zero, then the driver will update the value with the total size in bytes needed for all reports available. if size is non-zero, then driver will only retrieve the number of reports that fit into the buffer. if size is larger than size needed for all reports, then driver will update the value with the actual size needed.
- `pRawData`: [optional][range(0, *pRawDataSize)] buffer containing tracer reports in raw format

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricTracer`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

    - `nullptr == pRawDataSize`

zetMetricQueryPoolCreate

__ze_api_export *ze_result_t* __zecall **`zetMetricQueryPoolCreate`** (zet_device_handle_t *hDevice*, zet_metric_group_handle_t *hMetricGroup*, **`const`** *zet_metric_query_pool_desc_t* **desc*, zet_metric_query_pool_handle_t **phMetricQueryPool*)

Creates a pool of metric queries.

Parameters

- `hDevice`: handle of the device
- `hMetricGroup`: metric group associated with the query object.
- `desc`: metric query pool descriptor
- `phMetricQueryPool`: handle of metric query pool

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hDevice`
    - `nullptr == hMetricGroup`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == desc`
    - `nullptr == phMetricQueryPool`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
    - `ZET_METRIC_QUERY_POOL_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
    - desc->flags

zetMetricQueryPoolDestroy

__ze_api_export *ze_result_t* __zecall **zetMetricQueryPoolDestroy** (zet_metric_query_pool_handle_t *hMetricQueryPool*)

Deletes a query pool object.

Parameters

- `hMetricQueryPool`: [release] handle of the metric query pool

- The application is responsible for destroying all query handles created from the pool before destroying the pool itself
- The application is responsible for making sure the device is not currently referencing the any query within the pool before it is deleted
- The application may **not** call this function from simultaneous threads with the same query pool handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricQueryPool`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zetMetricQueryCreate

__ze_api_export *ze_result_t* __zecall **zetMetricQueryCreate** (zet_metric_query_pool_handle_t *hMetricQueryPool*, uint32_t *index*, zet_metric_query_handle_t **phMetricQuery*)

Creates metric query object.

Parameters

- `hMetricQueryPool`: handle of the metric query pool
- `index`: index of the query within the pool
- `phMetricQuery`: handle of metric query

- The application may call this function from simultaneous threads.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMetricQueryPool`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

    - `nullptr == phMetricQuery`

zetMetricQueryDestroy

__ze_api_export *ze_result_t* __zecall **`zetMetricQueryDestroy`**(zet_metric_query_handle_t *hMetricQuery*)

Deletes a metric query object.

Parameters

- `hMetricQuery`: [release] handle of metric query

- The application is responsible for making sure the device is not currently referencing the query before it is deleted
- The application may **not** call this function from simultaneous threads with the same query handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hMetricQuery`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zetMetricQueryReset

__ze_api_export *ze_result_t* __zecall **`zetMetricQueryReset`**(zet_metric_query_handle_t *hMetricQuery*)

Resets a metric query object back to inital state.

Parameters

- `hMetricQuery`: handle of metric query

- The application is responsible for making sure the device is not currently referencing the query before it is reset
- The application may **not** call this function from simultaneous threads with the same query handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hMetricQuery`

zetCommandListAppendMetricQueryBegin

__ze_api_export *ze_result_t* __zecall **zetCommandListAppendMetricQueryBegin**(zet_command_list_handle_t *hCommandList*, zet_metric_query_handle_t *hMetricQuery*)

Appends metric query begin into a command list.

Parameters

- `hCommandList`: handle of the command list
- `hMetricQuery`: handle of the metric query

- The application may **not** call this function from simultaneous threads with the same command list handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`
  - `nullptr == hMetricQuery`

zetCommandListAppendMetricQueryEnd

__ze_api_export *ze_result_t* __zecall **zetCommandListAppendMetricQueryEnd**(zet_command_list_handle_t *hCommandList*, zet_metric_query_handle_t *hMetricQuery*, ze_event_handle_t *hCompletionEvent*)

Appends metric query end into a command list.

Parameters

- `hCommandList`: handle of the command list
- `hMetricQuery`: handle of the metric query
- `hCompletionEvent`: [optional] handle of the completion event to signal

- The application may **not** call this function from simultaneous threads with the same command list handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hCommandList`

    - `nullptr == hMetricQuery`
  - *ZE_RESULT_ERROR_INVALID_SYNCHRONIZATION_OBJECT*

zetCommandListAppendMetricMemoryBarrier

__ze_api_export *ze_result_t* __zecall **`zetCommandListAppendMetricMemoryBarrier`** (zet_command_list_handle_t *hCommandList*)

Appends metric query commands to flush all caches.

Parameters

  - `hCommandList`: handle of the command list

- The application may **not** call this function from simultaneous threads with the same command list handle.

Return

  - *ZE_RESULT_SUCCESS*
  - *ZE_RESULT_ERROR_UNINITIALIZED*
  - *ZE_RESULT_ERROR_DEVICE_LOST*
  - *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hCommandList`

zetMetricQueryGetData

__ze_api_export *ze_result_t* __zecall **`zetMetricQueryGetData`** (zet_metric_query_handle_t *hMetricQuery*, size_t **pRawDataSize*, uint8_t **pRawData*)

Retrieves raw data for a given metric query.

Parameters

  - `hMetricQuery`: handle of the metric query
  - `pRawDataSize`: pointer to size in bytes of raw data requested to read. if size is zero, then the driver will update the value with the total size in bytes needed for all reports available. if size is non-zero, then driver will only retrieve the number of reports that fit into the buffer. if size is larger than size needed for all reports, then driver will update the value with the actual size needed.
  - `pRawData`: [optional][range(0, *pRawDataSize)] buffer containing query reports in raw format

- The application may call this function from simultaneous threads.

Return

  - *ZE_RESULT_SUCCESS*
  - *ZE_RESULT_ERROR_UNINITIALIZED*
  - *ZE_RESULT_ERROR_DEVICE_LOST*
  - *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hMetricQuery`

- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pRawDataSize`

20.3.2 Metric Enums

zet_metric_group_sampling_type_t

`enum zet_metric_group_sampling_type_t`
Metric group sampling type.

Values:

`enumerator ZET_METRIC_GROUP_SAMPLING_TYPE_EVENT_BASED` = ZE_BIT(0)
Event based sampling.

`enumerator ZET_METRIC_GROUP_SAMPLING_TYPE_TIME_BASED` = ZE_BIT(1)
Time based sampling.

zet_metric_group_properties_version_t

`enum zet_metric_group_properties_version_t`
API version of *zet_metric_group_properties_t*.

Values:

`enumerator ZET_METRIC_GROUP_PROPERTIES_VERSION_CURRENT` = ZE_MAKE_VERSION(0, 91)
version 0.91

zet_metric_type_t

`enum zet_metric_type_t`
Metric types.

Values:

`enumerator ZET_METRIC_TYPE_DURATION`
Metric type: duration.

`enumerator ZET_METRIC_TYPE_EVENT`
Metric type: event.

`enumerator ZET_METRIC_TYPE_EVENT_WITH_RANGE`
Metric type: event with range.

`enumerator ZET_METRIC_TYPE_THROUGHPUT`
Metric type: throughput.

`enumerator ZET_METRIC_TYPE_TIMESTAMP`
Metric type: timestamp.

`enumerator ZET_METRIC_TYPE_FLAG`
Metric type: flag.

`enumerator ZET_METRIC_TYPE_RATIO`
Metric type: ratio.

`enumerator ZET_METRIC_TYPE_RAW`
Metric type: raw.

zet_value_type_t

enum zet_value_type_t
Supported value types.

Values:

enumerator ZET_VALUE_TYPE_UINT32
32-bit unsigned-integer

enumerator ZET_VALUE_TYPE_UINT64
64-bit unsigned-integer

enumerator ZET_VALUE_TYPE_FLOAT32
32-bit floating-point

enumerator ZET_VALUE_TYPE_FLOAT64
64-bit floating-point

enumerator ZET_VALUE_TYPE_BOOL8
8-bit boolean

zet_metric_properties_version_t

enum zet_metric_properties_version_t
API version of *zet_metric_properties_t*.

Values:

enumerator ZET_METRIC_PROPERTIES_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

zet_metric_tracer_desc_version_t

enum zet_metric_tracer_desc_version_t
API version of *zet_metric_tracer_desc_t*.

Values:

enumerator ZET_METRIC_TRACER_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

zet_metric_query_pool_desc_version_t

enum zet_metric_query_pool_desc_version_t
API version of *zet_metric_query_pool_desc_t*.

Values:

enumerator ZET_METRIC_QUERY_POOL_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)
version 0.91

zet_metric_query_pool_flag_t

enum zet_metric_query_pool_flag_t

Metric query pool types.

Values:

enumerator ZET_METRIC_QUERY_POOL_FLAG_PERFORMANCE

Performance metric query pool.

enumerator ZET_METRIC_QUERY_POOL_FLAG_SKIP_EXECUTION

Skips workload execution between begin/end calls.

20.3.3 Metric Structures

zet_metric_group_properties_t

struct zet_metric_group_properties_t

Metric group properties queried using *zetMetricGroupGetProperties*.

Public Members

zet_metric_group_properties_version_t **version**

[in] *ZET_METRIC_GROUP_PROPERTIES_VERSION_CURRENT*

char **name**[**ZET_MAX_METRIC_GROUP_NAME**]

[out] metric group name

char **description**[**ZET_MAX_METRIC_GROUP_DESCRIPTION**]

[out] metric group description

zet_metric_group_sampling_type_t **samplingType**

[out] metric group sampling type

uint32_t **domain**

[out] metric group domain number. Cannot use simultaneous metric groups from different domains.

uint32_t **maxCommandQueueOrdinal**

[out] tracers and queries of this metric group cannot be submitted to a command queue with a larger ordinal value. See *ze_command_queue_desc_t* for more information on how to specify the command queue's ordinal.

uint32_t **metricCount**

[out] metric count belonging to this group

zet_value_t

union zet_value_t

#include <zet_metric.h> Union of values.

Public Members

uint32_t **ui32**
: [out] 32-bit unsigned-integer

uint64_t **ui64**
: [out] 32-bit unsigned-integer

float **fp32**
: [out] 32-bit floating-point

double **fp64**
: [out] 64-bit floating-point

ze_bool_t **b8**
: [out] 8-bit boolean

zet_typed_value_t

struct zet_typed_value_t
Typed value.

Public Members

zet_value_type_t **type**
: [out] type of value

zet_value_t **value**
: [out] value

zet_metric_properties_t

struct zet_metric_properties_t
Metric properties queried using *zetMetricGetProperties*.

Public Members

zet_metric_properties_version_t **version**
: [in] *ZET_METRIC_PROPERTIES_VERSION_CURRENT*

char **name**[**ZET_MAX_METRIC_NAME**]
: [out] metric name

char **description**[**ZET_MAX_METRIC_DESCRIPTION**]
: [out] metric description

char **component**[**ZET_MAX_METRIC_COMPONENT**]
: [out] metric component

uint32_t **tierNumber**
: [out] number of tier

zet_metric_type_t **metricType**
: [out] metric type

zet_value_type_t **resultType**
: [out] metric result type

char **resultUnits**[**ZET_MAX_METRIC_RESULT_UNITS**]
[out] metric result units

zet_metric_tracer_desc_t

struct zet_metric_tracer_desc_t
Metric tracer descriptor.

Public Members

zet_metric_tracer_desc_version_t **version**
[in] *ZET_METRIC_TRACER_DESC_VERSION_CURRENT*

uint32_t **notifyEveryNReports**
[in,out] number of collected reports after which notification event will be signalled

uint32_t **samplingPeriod**
[in,out] tracer sampling period in nanoseconds

zet_metric_query_pool_desc_t

struct zet_metric_query_pool_desc_t
Metric query pool description.

Public Members

zet_metric_query_pool_desc_version_t **version**
[in] *ZET_METRIC_QUERY_POOL_DESC_VERSION_CURRENT*

zet_metric_query_pool_flag_t **flags**
[in] Query pool type.

uint32_t **count**
[in] Internal slots count within query pool object.

20.4 Module

- Functions
  - *zetModuleGetDebugInfo*
- Enumerations
  - *zet_module_debug_info_format_t*

20.4.1 Module Functions

zetModuleGetDebugInfo

__ze_api_export *ze_result_t* __zecall **zetModuleGetDebugInfo**(zet_module_handle_t *hModule*, *zet_module_debug_info_format_t format*, size_t **pSize*, uint8_t **pDebugInfo*)

Retrieve debug info from module.

Parameters

- `hModule`: handle of the module
- `format`: debug info format requested
- `pSize`: size of debug info in bytes
- `pDebugInfo`: [optional] byte pointer to debug info

- The caller can pass nullptr for pDebugInfo when querying only for size.
- The implementation will copy the native binary into a buffer supplied by the caller.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hModule`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - format
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pSize`

20.4.2 Module Enums

zet_module_debug_info_format_t

enum zet_module_debug_info_format_t

Supported module debug info formats.

Values:

enumerator ZET_MODULE_DEBUG_INFO_FORMAT_ELF_DWARF

Format is ELF/DWARF.

20.5 Pin

- Functions
  - *zetKernelGetProfileInfo*
- Enumerations
  - *zet_profile_info_version_t*
  - *zet_profile_flag_t*
  - *zet_profile_token_type_t*
- Structures
  - *zet_profile_info_t*
  - *zet_profile_free_register_token_t*
  - *zet_profile_register_sequence_t*

20.5.1 Pin Functions

zetKernelGetProfileInfo

__ze_api_export *ze_result_t* __zecall **`zetKernelGetProfileInfo`**(zet_kernel_handle_t *hKernel*, *zet_profile_info_t* **pInfo*)

Retrieve profiling information generated for the kernel.

Parameters

- `hKernel`: handle to kernel
- `pInfo`: pointer to profile info

- Module must be created using the following build option:
  - ”-zet-profile-flags <n>” - enable generation of profile information
  - ”<n>” must be a combination of *zet_profile_flag_t*, in hex
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hKernel`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pInfo`

20.5.2 Pin Enums

zet_profile_info_version_t

enum zet_profile_info_version_t

API version of *zet_profile_info_t*.

Values:

enumerator ZET_PROFILE_INFO_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

zet_profile_flag_t

enum zet_profile_flag_t

Supportted profile features.

Values:

enumerator ZET_PROFILE_FLAG_REGISTER_REALLOCATION = ZE_BIT(0)

possible to allow for instrumentation

request the compiler attempt to minimize register usage as much as

enumerator ZET_PROFILE_FLAG_FREE_REGISTER_INFO = ZE_BIT(1)

request the compiler generate free register info

zet_profile_token_type_t

enum zet_profile_token_type_t

Supported profile token types.

Values:

enumerator ZET_PROFILE_TOKEN_TYPE_FREE_REGISTER

GRF info.

20.5.3 Pin Structures

zet_profile_info_t

struct zet_profile_info_t

Profiling meta-data for instrumentation.

Public Members

zet_profile_info_version_t **version**

[in] *ZET_PROFILE_INFO_VERSION_CURRENT*

zet_profile_flag_t **flags**

[out] indicates which flags were enabled during compilation

uint32_t **numTokens**

[out] number of tokens immediately following this structure

zet_profile_free_register_token_t

struct zet_profile_free_register_token_t

Profile free register token detailing unused registers in the current function.

Public Members

zet_profile_token_type_t **type**

[out] type of token

uint32_t **size**

[out] total size of the token, in bytes

uint32_t **count**

[out] number of register sequences immediately following this structure

zet_profile_register_sequence_t

struct zet_profile_register_sequence_t

Profile register sequence detailing consecutive bytes, all of which are unused.

Public Members

uint32_t **start**

[out] starting byte in the register table, representing the start of unused bytes in the current function

uint32_t **count**

[out] number of consecutive bytes in the sequence, starting from start

20.6 Sysman

- Functions
  - *zetSysmanGet*
  - *zetSysmanDeviceGetProperties*
  - *zetSysmanSchedulerGetSupportedModes*
  - *zetSysmanSchedulerGetCurrentMode*
  - *zetSysmanSchedulerGetTimeoutModeProperties*
  - *zetSysmanSchedulerGetTimesliceModeProperties*
  - *zetSysmanSchedulerSetTimeoutMode*
  - *zetSysmanSchedulerSetTimesliceMode*
  - *zetSysmanSchedulerSetExclusiveMode*
  - *zetSysmanSchedulerSetComputeUnitDebugMode*
  - *zetSysmanPerformanceProfileGetSupported*
  - *zetSysmanPerformanceProfileGet*
  - *zetSysmanPerformanceProfileSet*

- *zetSysmanProcessesGetState*
- *zetSysmanDeviceReset*
- *zetSysmanDeviceGetRepairStatus*
- *zetSysmanPciGetProperties*
- *zetSysmanPciGetState*
- *zetSysmanPciGetBars*
- *zetSysmanPciGetStats*
- *zetSysmanPowerGet*
- *zetSysmanPowerGetProperties*
- *zetSysmanPowerGetEnergyCounter*
- *zetSysmanPowerGetLimits*
- *zetSysmanPowerSetLimits*
- *zetSysmanPowerGetEnergyThreshold*
- *zetSysmanPowerSetEnergyThreshold*
- *zetSysmanFrequencyGet*
- *zetSysmanFrequencyGetProperties*
- *zetSysmanFrequencyGetAvailableClocks*
- *zetSysmanFrequencyGetRange*
- *zetSysmanFrequencySetRange*
- *zetSysmanFrequencyGetState*
- *zetSysmanFrequencyGetThrottleTime*
- *zetSysmanFrequencyOcGetCapabilities*
- *zetSysmanFrequencyOcGetConfig*
- *zetSysmanFrequencyOcSetConfig*
- *zetSysmanFrequencyOcGetIccMax*
- *zetSysmanFrequencyOcSetIccMax*
- *zetSysmanFrequencyOcGetTjMax*
- *zetSysmanFrequencyOcSetTjMax*
- *zetSysmanEngineGet*
- *zetSysmanEngineGetProperties*
- *zetSysmanEngineGetActivity*
- *zetSysmanStandbyGet*
- *zetSysmanStandbyGetProperties*
- *zetSysmanStandbyGetMode*
- *zetSysmanStandbySetMode*
- *zetSysmanFirmwareGet*

- *zetSysmanFirmwareGetProperties*
- *zetSysmanFirmwareGetChecksum*
- *zetSysmanFirmwareFlash*
- *zetSysmanMemoryGet*
- *zetSysmanMemoryGetProperties*
- *zetSysmanMemoryGetState*
- *zetSysmanMemoryGetBandwidth*
- *zetSysmanFabricPortGet*
- *zetSysmanFabricPortGetProperties*
- *zetSysmanFabricPortGetLinkType*
- *zetSysmanFabricPortGetConfig*
- *zetSysmanFabricPortSetConfig*
- *zetSysmanFabricPortGetState*
- *zetSysmanFabricPortGetThroughput*
- *zetSysmanTemperatureGet*
- *zetSysmanTemperatureGetProperties*
- *zetSysmanTemperatureGetConfig*
- *zetSysmanTemperatureSetConfig*
- *zetSysmanTemperatureGetState*
- *zetSysmanPsuGet*
- *zetSysmanPsuGetProperties*
- *zetSysmanPsuGetState*
- *zetSysmanFanGet*
- *zetSysmanFanGetProperties*
- *zetSysmanFanGetConfig*
- *zetSysmanFanSetConfig*
- *zetSysmanFanGetState*
- *zetSysmanLedGet*
- *zetSysmanLedGetProperties*
- *zetSysmanLedGetState*
- *zetSysmanLedSetState*
- *zetSysmanRasGet*
- *zetSysmanRasGetProperties*
- *zetSysmanRasGetConfig*
- *zetSysmanRasSetConfig*
- *zetSysmanRasGetState*

  - *zetSysmanEventGet*
  - *zetSysmanEventGetConfig*
  - *zetSysmanEventSetConfig*
  - *zetSysmanEventGetState*
  - *zetSysmanEventListen*
  - *zetSysmanDiagnosticsGet*
  - *zetSysmanDiagnosticsGetProperties*
  - *zetSysmanDiagnosticsGetTests*
  - *zetSysmanDiagnosticsRunTests*
- Enumerations
  - *zet_sysman_version_t*
  - *zet_engine_type_t*
  - *zet_sched_mode_t*
  - *zet_perf_profile_t*
  - *zet_repair_status_t*
  - *zet_pci_link_status_t*
  - *zet_pci_link_qual_issues_t*
  - *zet_pci_link_stab_issues_t*
  - *zet_pci_bar_type_t*
  - *zet_freq_domain_t*
  - *zet_freq_throttle_reasons_t*
  - *zet_oc_mode_t*
  - *zet_engine_group_t*
  - *zet_standby_type_t*
  - *zet_standby_promo_mode_t*
  - *zet_mem_type_t*
  - *zet_mem_health_t*
  - *zet_fabric_port_status_t*
  - *zet_fabric_port_qual_issues_t*
  - *zet_fabric_port_stab_issues_t*
  - *zet_temp_sensors_t*
  - *zet_psu_voltage_status_t*
  - *zet_fan_speed_mode_t*
  - *zet_fan_speed_units_t*
  - *zet_ras_error_type_t*
  - *zet_sysman_event_type_t*

  - *zet_diag_type_t*
  - *zet_diag_result_t*
- Structures
  - *zet_sysman_properties_t*
  - *zet_sched_timeout_properties_t*
  - *zet_sched_timeslice_properties_t*
  - *zet_process_state_t*
  - *zet_pci_address_t*
  - *zet_pci_speed_t*
  - *zet_pci_properties_t*
  - *zet_pci_state_t*
  - *zet_pci_bar_properties_t*
  - *zet_pci_stats_t*
  - *zet_power_properties_t*
  - *zet_power_energy_counter_t*
  - *zet_power_sustained_limit_t*
  - *zet_power_burst_limit_t*
  - *zet_power_peak_limit_t*
  - *zet_energy_threshold_t*
  - *zet_freq_properties_t*
  - *zet_freq_range_t*
  - *zet_freq_state_t*
  - *zet_freq_throttle_time_t*
  - *zet_oc_capabilities_t*
  - *zet_oc_config_t*
  - *zet_engine_properties_t*
  - *zet_engine_stats_t*
  - *zet_standby_properties_t*
  - *zet_firmware_properties_t*
  - *zet_mem_properties_t*
  - *zet_mem_state_t*
  - *zet_mem_bandwidth_t*
  - *zet_fabric_port_uuid_t*
  - *zet_fabric_port_speed_t*
  - *zet_fabric_port_properties_t*
  - *zet_fabric_link_type_t*

- *zet_fabric_port_config_t*
- *zet_fabric_port_state_t*
- *zet_fabric_port_throughput_t*
- *zet_temp_properties_t*
- *zet_temp_threshold_t*
- *zet_temp_config_t*
- *zet_psu_properties_t*
- *zet_psu_state_t*
- *zet_fan_temp_speed_t*
- *zet_fan_properties_t*
- *zet_fan_config_t*
- *zet_led_properties_t*
- *zet_led_state_t*
- *zet_ras_properties_t*
- *zet_ras_details_t*
- *zet_ras_config_t*
- *zet_event_config_t*
- *zet_diag_test_t*
- *zet_diag_properties_t*

20.6.1 Sysman Functions

zetSysmanGet

__ze_api_export *ze_result_t* __zecall **zetSysmanGet**(zet_device_handle_t *hDevice*, *zet_sysman_version_t* *version*, zet_sysman_handle_t **phSysman*)

Get the handle to access Sysman features for a device.

Parameters

- `hDevice`: Handle of the device
- `version`: Sysman version that application was built with
- `phSysman`: Handle for accessing Sysman features

- The returned handle is unique.
- zet_device_handle_t returned by *zeDeviceGetSubDevices()* are not support. Only use handles returned by *zeDeviceGet()*. All resources on sub-devices can be enumerated through the primary device.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*

- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDevice`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - version
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phSysman`

zetSysmanDeviceGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanDeviceGetProperties**(zet_sysman_handle_t *hSysman*, *zet_sysman_properties_t* **pProperties*)

Get properties about the device.

Parameters

- `hSysman`: Sysman handle of the device.
- `pProperties`: Structure that will contain information about the device.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanSchedulerGetSupportedModes

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerGetSupportedModes**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, *zet_sched_mode_t* **pModes*)

Get a list of supported scheduler modes.

Parameters

- `hSysman`: Sysman handle of the device.

    - `pCount`: pointer to the number of scheduler modes. if count is zero, then the driver will update the value with the total number of supported modes. if count is non-zero, then driver will only retrieve that number of supported scheduler modes. if count is larger than the number of supported scheduler modes, then the driver will update the value with the correct number of supported scheduler modes that are returned.
    - `pModes`: [optional][range(0, *pCount)] Array of supported scheduler modes

- If zero modes are returned, control of scheduler modes are not supported.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanSchedulerGetCurrentMode

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerGetCurrentMode**(zet_sysman_handle_t *hSysman*, *zet_sched_mode_t* **pMode*)

Get current scheduler mode.

Parameters

- `hSysman`: Sysman handle of the device.
- `pMode`: Will contain the current scheduler mode.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pMode`

- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Device does not support scheduler modes (check using *zetSysmanSchedulerGetSupportedModes()*).

zetSysmanSchedulerGetTimeoutModeProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerGetTimeoutModeProperties**(zet_sysman_handle_t *hSysman*, ze_bool_t *getDefaults*, *zet_sched_timeout_properties_t* **pConfig*)

Get scheduler config for mode *ZET_SCHED_MODE_TIMEOUT*.

Parameters

- `hSysman`: Sysman handle of the device.
- `getDefaults`: If TRUE, the driver will return the system default properties for this mode, otherwise it will return the current properties.
- `pConfig`: Will contain the current parameters for this mode.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).

zetSysmanSchedulerGetTimesliceModeProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerGetTimesliceModeProperties**(zet_sysman_handle_t *hSysman*, ze_bool_t *getDefaults*, *zet_sched_timeslice_propertie* **pConfig*)

Get scheduler config for mode *ZET_SCHED_MODE_TIMESLICE*.

Parameters

- `hSysman`: Sysman handle of the device.
- `getDefaults`: If TRUE, the driver will return the system default properties for this mode, otherwise it will return the current properties.
- `pConfig`: Will contain the current parameters for this mode.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).

zetSysmanSchedulerSetTimeoutMode

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerSetTimeoutMode**(zet_sysman_handle_t *hSysman*, *zet_sched_timeout_properties_t* **pProperties*, ze_bool_t **pNeedReboot*)

Change scheduler mode to *ZET_SCHED_MODE_TIMEOUT* or update scheduler mode parameters if already running in this mode.

Parameters

- `hSysman`: Sysman handle of the device.

- `pProperties`: The properties to use when configurating this mode.
- `pNeedReboot`: Will be set to TRUE if a system reboot is needed to apply the new scheduler mode.

- This mode is optimized for multiple applications or contexts submitting work to the hardware. When higher priority work arrives, the scheduler attempts to pause the current executing work within some timeout interval, then submits the other work.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`
  - `nullptr == pNeedReboot`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make this modification.

zetSysmanSchedulerSetTimesliceMode

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerSetTimesliceMode**(zet_sysman_handle_t *hSysman*, *zet_sched_timeslice_properties_t* **pProperties*, ze_bool_t **pNeedReboot*)

Change scheduler mode to *ZET_SCHED_MODE_TIMESLICE* or update scheduler mode parameters if already running in this mode.

Parameters

- `hSysman`: Sysman handle of the device.
- `pProperties`: The properties to use when configurating this mode.
- `pNeedReboot`: Will be set to TRUE if a system reboot is needed to apply the new scheduler mode.

- This mode is optimized to provide fair sharing of hardware execution time between multiple contexts submitting work to the hardware concurrently.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`
  - `nullptr == pNeedReboot`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make this modification.

zetSysmanSchedulerSetExclusiveMode

__ze_api_export *ze_result_t* __zecall **`zetSysmanSchedulerSetExclusiveMode`** (zet_sysman_handle_t *hSysman*, ze_bool_t **pNeedReboot*)

Change scheduler mode to *ZET_SCHED_MODE_EXCLUSIVE*.

Parameters

- `hSysman`: Sysman handle of the device.
- `pNeedReboot`: Will be set to TRUE if a system reboot is needed to apply the new scheduler mode.

- This mode is optimized for single application/context use-cases. It permits a context to run indefinitely on the hardware without being preempted or terminated. All pending work for other contexts must wait until the running context completes with no further submitted work.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pNeedReboot`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).

- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make this modification.

zetSysmanSchedulerSetComputeUnitDebugMode

__ze_api_export *ze_result_t* __zecall **zetSysmanSchedulerSetComputeUnitDebugMode**(zet_sysman_handle_t *hSysman*, ze_bool_t **pNeedReboot*)

Change scheduler mode to *ZET_SCHED_MODE_COMPUTE_UNIT_DEBUG*.

Parameters

- `hSysman`: Sysman handle of the device.
- `pNeedReboot`: Will be set to TRUE if a system reboot is needed to apply the new scheduler mode.

- This mode is optimized for application debug. It ensures that only one command queue can execute work on the hardware at a given time. Work is permitted to run as long as needed without enforcing any scheduler fairness policies.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pNeedReboot`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - This scheduler mode is not supported (check using *zetSysmanSchedulerGetSupportedModes()*).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make this modification.

zetSysmanPerformanceProfileGetSupported

__ze_api_export *ze_result_t* __zecall **zetSysmanPerformanceProfileGetSupported**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, *zet_perf_profile_t* **pProfiles*)

Get a list of supported performance profiles that can be loaded for this device.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of performance profiles. if count is zero, then the driver will update the value with the total number of supported performance profiles. if count is non-zero, then driver will only retrieve that number of supported performance profiles. if count is larger than the number of supported performance profiles, then the driver will update the value with the correct number of supported performance profiles that are returned.
- `pProfiles`: [optional][range(0, *pCount)] Array of supported performance profiles

- The balanced profile *ZET_PERF_PROFILE_BALANCED* is always returned in the array.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanPerformanceProfileGet

__ze_api_export *ze_result_t* __zecall **zetSysmanPerformanceProfileGet**(zet_sysman_handle_t *hSysman*, *zet_perf_profile_t* **pProfile*)

Get current pre-configured performance profile being used by the hardware.

Parameters

- `hSysman`: Sysman handle of the device.
- `pProfile`: The performance profile currently loaded.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProfile`

zetSysmanPerformanceProfileSet

__ze_api_export *ze_result_t* __zecall **`zetSysmanPerformanceProfileSet`** (zet_sysman_handle_t *hSysman*, *zet_perf_profile_t profile*)

Load a pre-configured performance profile.

Parameters

- `hSysman`: Sysman handle of the device.
- `profile`: The performance profile to load.

- Performance profiles are not persistent settings. If the device is reset, the device will default back to the balanced profile *ZET_PERF_PROFILE_BALANCED*.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - profile
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - The specified profile is not valid or not supported on this device (use *zetSysmanPerformanceProfileGetSupported()* to get a list of supported profiles).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to change the performance profile of the hardware.

zetSysmanProcessesGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanProcessesGetState**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, *zet_process_state_t* **pProcesses*)

Get information about host processes using the device.

Parameters

- `hSysman`: Sysman handle for the device
- `pCount`: pointer to the number of processes. if count is zero, then the driver will update the value with the total number of processes currently using the device. if count is non-zero but less than the number of processes, the driver will set to the number of processes currently using the device and return the error *ZE_RESULT_ERROR_INVALID_SIZE*. if count is larger than the number of processes, then the driver will update the value with the correct number of processes that are returned.
- `pProcesses`: [optional][range(0, *pCount)] array of process information, one for each process currently using the device

- The number of processes connected to the device is dynamic. This means that between a call to determine the correct value of pCount and the subsequent call, the number of processes may have increased. It is recommended that a large array be passed in so as to avoid receiving the error *ZE_RESULT_ERROR_INVALID_SIZE*.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`
- *ZE_RESULT_ERROR_INVALID_SIZE*
  - The provided value of pCount is not big enough to store information about all the processes currently attached to the device.

zetSysmanDeviceReset

__ze_api_export *ze_result_t* __zecall **zetSysmanDeviceReset**(zet_sysman_handle_t *hSysman*)

Reset device.

Parameters

- `hSysman`: Sysman handle for the device

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to perform this operation.

zetSysmanDeviceGetRepairStatus

__ze_api_export *ze_result_t* __zecall **zetSysmanDeviceGetRepairStatus**(zet_sysman_handle_t *hSysman*, *zet_repair_status_t* **pRepairStatus*)

Find out if the device has been repaired (either by the manufacturer or by running diagnostics)

Parameters

- `hSysman`: Sysman handle for the device
- `pRepairStatus`: Will indicate if the device was repaired

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pRepairStatus`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to query this property.

zetSysmanPciGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanPciGetProperties**(zet_sysman_handle_t *hSysman*, *zet_pci_properties_t* **pProperties*)

Get PCI properties - address, max speed.

Parameters

- `hSysman`: Sysman handle of the device.
- `pProperties`: Will contain the PCI properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanPciGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanPciGetState**(zet_sysman_handle_t *hSysman*, *zet_pci_state_t* **pState*)

Get current PCI state - current speed.

Parameters

- `hSysman`: Sysman handle of the device.
- `pState`: Will contain the PCI properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`

zetSysmanPciGetBars

__ze_api_export *ze_result_t* __zecall **zetSysmanPciGetBars**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, *zet_pci_bar_properties_t* **pProperties*)

Get information about each configured bar.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of PCI bars. if count is zero, then the driver will update the value with the total number of bars. if count is non-zero, then driver will only retrieve that number of bars. if count is larger than the number of bar, then the driver will update the value with the correct number of bars that are returned.
- `pProperties`: [optional][range(0, *pCount)] array of bar properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanPciGetStats

__ze_api_export *ze_result_t* __zecall **zetSysmanPciGetStats**(zet_sysman_handle_t *hSysman*, *zet_pci_stats_t* **pStats*)

Get PCI stats - bandwidth, number of packets, number of replays.

Parameters

- `hSysman`: Sysman handle of the device.
- `pStats`: Will contain a snapshot of the latest stats.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pStats`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to query this telemetry.

zetSysmanPowerGet

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_pwr_handle_t **phPower*)

Get handle of power domains.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phPower`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanPowerGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerGetProperties** (zet_sysman_pwr_handle_t *hPower*, *zet_power_properties_t* **pProperties*)

Get properties related to a power domain.

Parameters

- `hPower`: Handle for the component.
- `pProperties`: Structure that will contain property data.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanPowerGetEnergyCounter

__ze_api_export *ze_result_t* __zecall **`zetSysmanPowerGetEnergyCounter`**(zet_sysman_pwr_handle_t *hPower*, *zet_power_energy_counter_t* **pEnergy*)

Get energy counter.

Parameters

- `hPower`: Handle for the component.
- `pEnergy`: Will contain the latest snapshot of the energy counter and timestamp when the last counter value was measured.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pEnergy`

zetSysmanPowerGetLimits

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerGetLimits**(zet_sysman_pwr_handle_t *hPower*, *zet_power_sustained_limit_t* **pSustained*, *zet_power_burst_limit_t* **pBurst*, *zet_power_peak_limit_t* **pPeak*)

Get power limits.

Parameters

- `hPower`: Handle for the component.
- `pSustained`: [optional] The sustained power limit.
- `pBurst`: [optional] The burst power limit.
- `pPeak`: [optional] The peak power limit.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`

zetSysmanPowerSetLimits

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerSetLimits**(zet_sysman_pwr_handle_t *hPower*, **const** *zet_power_sustained_limit_t* **pSustained*, **const** *zet_power_burst_limit_t* **pBurst*, **const** *zet_power_peak_limit_t* **pPeak*)

Set power limits.

Parameters

- `hPower`: Handle for the component.
- `pSustained`: [optional] The sustained power limit.
- `pBurst`: [optional] The burst power limit.
- `pPeak`: [optional] The peak power limit.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*

- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - The device is in use, meaning that the GPU is under Over clocking, applying power limits under overclocking is not supported.

zetSysmanPowerGetEnergyThreshold

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerGetEnergyThreshold**(zet_sysman_pwr_handle_t *hPower*, *zet_energy_threshold_t* **pThreshold*)

Get energy threshold.

Parameters

- `hPower`: Handle for the component.
- `pThreshold`: Returns information about the energy threshold setting - enabled/energy threshold/process ID.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pThreshold`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Energy threshold not supported on this power domain (check *zet_power_properties_t.isEnergyThresholdSupported*).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to request this feature.

zetSysmanPowerSetEnergyThreshold

__ze_api_export *ze_result_t* __zecall **zetSysmanPowerSetEnergyThreshold**(zet_sysman_pwr_handle_t *hPower*, double *threshold*)

Set energy threshold.

Parameters

- `hPower`: Handle for the component.
- `threshold`: The energy threshold to be set in joules.

- An event *ZET_SYSMAN_EVENT_TYPE_ENERGY_THRESHOLD_CROSSED* will be generated when the delta energy consumed starting from this call exceeds the specified threshold. Use the function *zetSysmanEventSetConfig()* to start receiving the event.
- Only one running process can control the energy threshold at a given time. If another process attempts to change the energy threshold, the error *ZE_RESULT_ERROR_NOT_AVAILABLE* will be returned. The function *zetSysmanPowerGetEnergyThreshold()* to determine the process ID currently controlling this setting.
- Calling this function will remove any pending energy thresholds and start counting from the time of this call.
- Once the energy threshold has been reached and the event generated, the threshold is automatically removed. It is up to the application to request a new threshold.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPower`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Energy threshold not supported on this power domain (check *zet_power_properties_t.isEnergyThresholdSupported*).
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to request this feature.
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - Another running process has set the energy threshold.

zetSysmanFrequencyGet

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_freq_handle_t **phFrequency*)

Get handle of frequency domains.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phFrequency`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanFrequencyGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGetProperties** (zet_sysman_freq_handle_t *hFrequency*, *zet_freq_properties_t* **pProperties*)

Get frequency properties - available frequencies.

Parameters

- `hFrequency`: Handle for the component.
- `pProperties`: The frequency properties for the specified domain.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanFrequencyGetAvailableClocks

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGetAvailableClocks**(zet_sysman_freq_handle_t *hFrequency*, uint32_t **pCount*, double **phFrequency*)

Get available non-overclocked hardware clock frequencies for the frequency domain.

Parameters

- `hFrequency`: Sysman handle of the device.
- `pCount`: pointer to the number of frequencies. If count is zero, then the driver will update the value with the total number of frequencies available. If count is non-zero, then driver will only retrieve that number of frequencies. If count is larger than the number of frequencies available, then the driver will update the value with the correct number of frequencies available.
- `phFrequency`: [optional][range(0, *pCount)] array of frequencies in units of MHz and sorted from slowest to fastest

- The list of available frequencies is returned in order of slowest to fastest.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanFrequencyGetRange

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGetRange**(zet_sysman_freq_handle_t *hFrequency*, *zet_freq_range_t* **pLimits*)

Get current frequency limits.

Parameters

- `hFrequency`: Handle for the component.
- `pLimits`: The range between which the hardware can operate for the specified domain.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pLimits`

zetSysmanFrequencySetRange

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencySetRange**(zet_sysman_freq_handle_t *hFrequency*, **const** *zet_freq_range_t* **pLimits*)

Set frequency range between which the hardware can operate.

Parameters

- `hFrequency`: Handle for the component.
- `pLimits`: The limits between which the hardware can operate for the specified domain.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

    - `nullptr == pLimits`
  - *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
    - User does not have permissions to make these modifications.

zetSysmanFrequencyGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGetState**(zet_sysman_freq_handle_t *hFrequency*, *zet_freq_state_t *pState*)

Get current frequency state - frequency request, actual frequency, TDP limits.

Parameters

  - `hFrequency`: Handle for the component.
  - `pState`: Frequency state for the specified domain.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

  - *ZE_RESULT_SUCCESS*
  - *ZE_RESULT_ERROR_UNINITIALIZED*
  - *ZE_RESULT_ERROR_DEVICE_LOST*
  - *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hFrequency`
  - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pState`

zetSysmanFrequencyGetThrottleTime

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyGetThrottleTime**(zet_sysman_freq_handle_t *hFrequency*, *zet_freq_throttle_time_t *pThrottleTime*)

Get frequency throttle time.

Parameters

  - `hFrequency`: Handle for the component.
  - `pThrottleTime`: Will contain a snapshot of the throttle time counters for the specified domain.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

  - *ZE_RESULT_SUCCESS*

- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pThrottleTime`

zetSysmanFrequencyOcGetCapabilities

__ze_api_export *ze_result_t* __zecall **`zetSysmanFrequencyOcGetCapabilities`** (zet_sysman_freq_handle_t *hFrequency*, *zet_oc_capabilities_t* **pOcCapabilities*)

Get the overclocking capabilities.

Parameters

- `hFrequency`: Handle for the component.
- `pOcCapabilities`: Pointer to the capabilities structure *zet_oc_capabilities_t*.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pOcCapabilities`

zetSysmanFrequencyOcGetConfig

__ze_api_export *ze_result_t* __zecall **`zetSysmanFrequencyOcGetConfig`** (zet_sysman_freq_handle_t *hFrequency*, *zet_oc_config_t* **pOcConfiguration*)

Get the current overclocking configuration.

Parameters

- `hFrequency`: Handle for the component.
- `pOcConfiguration`: Pointer to the configuration structure *zet_oc_config_t*.

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pOcConfiguration`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)

zetSysmanFrequencyOcSetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyOcSetConfig**(zet_sysman_freq_handle_t *hFrequency*, *zet_oc_config_t* **pOcConfiguration*, ze_bool_t **pDeviceRestart*)

Change the overclocking configuration.

Parameters

- `hFrequency`: Handle for the component.
- `pOcConfiguration`: Pointer to the configuration structure *zet_oc_config_t*.
- `pDeviceRestart`: This will be set to true if the device needs to be restarted in order to enable the new overclock settings.

- If *zet_oc_config_t.mode* is set to *ZET_OC_MODE_OFF*, overclocking will be turned off and the hardware returned to run with factory voltages/frequencies. Call *zetSysmanFrequencyOcSetIccMax()* and *zetSysmanFrequencyOcSetTjMax()* separately with 0.0 to return those settings to factory defaults.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pOcConfiguration`

    - `nullptr == pDeviceRestart`

- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
    - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)
    - The specified voltage and/or frequency overclock settings exceed the hardware values (see *zet_oc_capabilities_t.maxOcFrequency*, *zet_oc_capabilities_t.maxOcVoltage*, *zet_oc_capabilities_t.minOcVoltageOffset*, *zet_oc_capabilities_t.maxOcVoltageOffset*).
    - Requested voltage overclock is very high but *zet_oc_capabilities_t.isHighVoltModeEnabled* is not enabled for the device.
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
    - Overclocking feature is locked on this frequency domain
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
    - User does not have permissions to make these modifications.

zetSysmanFrequencyOcGetIccMax

__ze_api_export *ze_result_t* __zecall **`zetSysmanFrequencyOcGetIccMax`**(zet_sysman_freq_handle_t *hFrequency*, double **pOcIccMax*)

Get the maximum current limit setting.

Parameters

- `hFrequency`: Handle for the component.
- `pOcIccMax`: Will contain the maximum current limit in Amperes on successful return.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
    - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pOcIccMax`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
    - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)
    - Capability *zet_oc_capabilities_t.isIccMaxSupported* is false for this frequency domain

zetSysmanFrequencyOcSetIccMax

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyOcSetIccMax**(zet_sysman_freq_handle_t *hFrequency*, double *ocIccMax*)

Change the maximum current limit setting.

Parameters

- `hFrequency`: Handle for the component.
- `ocIccMax`: The new maximum current limit in Amperes.

- Setting ocIccMax to 0.0 will return the value to the factory default.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)
  - Capability *zet_oc_capabilities_t.isIccMaxSupported* is false for this frequency domain
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - Overclocking feature is locked on this frequency domain
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - The specified current limit is too low or too high
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanFrequencyOcGetTjMax

__ze_api_export *ze_result_t* __zecall **zetSysmanFrequencyOcGetTjMax**(zet_sysman_freq_handle_t *hFrequency*, double **pOcTjMax*)

Get the maximum temperature limit setting.

Parameters

- `hFrequency`: Handle for the component.
- `pOcTjMax`: Will contain the maximum temperature limit in degrees Celsius on successful return.

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pOcTjMax`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)

zetSysmanFrequencyOcSetTjMax

__ze_api_export *ze_result_t* __zecall **`zetSysmanFrequencyOcSetTjMax`** (zet_sysman_freq_handle_t *hFrequency*, double *ocTjMax*)

Change the maximum temperature limit setting.

Parameters

- `hFrequency`: Handle for the component.
- `ocTjMax`: The new maximum temperature limit in degrees Celsius.

- Setting ocTjMax to 0.0 will return the value to the factory default.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFrequency`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Overclocking is not supported on this frequency domain (*zet_oc_capabilities_t.isOcSupported*)
  - Capability *zet_oc_capabilities_t.isTjMaxSupported* is false for this frequency domain
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - Overclocking feature is locked on this frequency domain
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - The specified temperature limit is too high

- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanEngineGet

__ze_api_export *ze_result_t* __zecall **zetSysmanEngineGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_engine_handle_t **phEngine*)

Get handle of engine groups.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phEngine`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanEngineGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanEngineGetProperties** (zet_sysman_engine_handle_t *hEngine*, *zet_engine_properties_t* **pProperties*)

Get engine group properties.

Parameters

- `hEngine`: Handle for the component.
- `pProperties`: The properties for the specified engine group.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEngine`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanEngineGetActivity

__ze_api_export *ze_result_t* __zecall **zetSysmanEngineGetActivity**(zet_sysman_engine_handle_t *hEngine*, *zet_engine_stats_t* **pStats*)

Get the activity stats for an engine group.

Parameters

- `hEngine`: Handle for the component.
- `pStats`: Will contain a snapshot of the engine group activity counters.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEngine`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pStats`

zetSysmanStandbyGet

__ze_api_export *ze_result_t* __zecall **zetSysmanStandbyGet**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_standby_handle_t **phStandby*)

Get handle of standby controls.

Parameters

- `hSysman`: Sysman handle of the device.

- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phStandby`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanStandbyGetProperties

__ze_api_export *ze_result_t* __zecall **`zetSysmanStandbyGetProperties`** (zet_sysman_standby_handle_t *hStandby*, *zet_standby_properties_t* **pProperties*)

Get standby hardware component properties.

Parameters

- `hStandby`: Handle for the component.
- `pProperties`: Will contain the standby hardware properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hStandby`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanStandbyGetMode

__ze_api_export *ze_result_t* __zecall **zetSysmanStandbyGetMode**(zet_sysman_standby_handle_t *hStandby*, *zet_standby_promo_mode_t* **pMode*)

Get the current standby promotion mode.

Parameters

- `hStandby`: Handle for the component.
- `pMode`: Will contain the current standby mode.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hStandby`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pMode`

zetSysmanStandbySetMode

__ze_api_export *ze_result_t* __zecall **zetSysmanStandbySetMode**(zet_sysman_standby_handle_t *hStandby*, *zet_standby_promo_mode_t* *mode*)

Set standby promotion mode.

Parameters

- `hStandby`: Handle for the component.
- `mode`: New standby mode.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hStandby`

- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
  - mode
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanFirmwareGet

__ze_api_export *ze_result_t* __zecall **zetSysmanFirmwareGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_firmware_handle_t **phFirmware*)

Get handle of firmwares.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phFirmware`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanFirmwareGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanFirmwareGetProperties** (zet_sysman_firmware_handle_t *hFirmware*, *zet_firmware_properties_t* **pProperties*)

Get firmware properties.

Parameters

- `hFirmware`: Handle for the component.
- `pProperties`: Pointer to an array that will hold the properties of the firmware

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFirmware`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanFirmwareGetChecksum

__ze_api_export *ze_result_t* __zecall **`zetSysmanFirmwareGetChecksum`**(zet_sysman_firmware_handle_t *hFirmware*, uint32_t **pChecksum*)

Get firmware checksum.

Parameters

- `hFirmware`: Handle for the component.
- `pChecksum`: Calculated checksum of the installed firmware.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFirmware`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pChecksum`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to perform this operation.

zetSysmanFirmwareFlash

__ze_api_export *ze_result_t* __zecall **zetSysmanFirmwareFlash**(zet_sysman_firmware_handle_t *hFirmware*, void **pImage*, uint32_t *size*)

Flash a new firmware image.

Parameters

- `hFirmware`: Handle for the component.
- `pImage`: Image of the new firmware to flash.
- `size`: Size of the flash image.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFirmware`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pImage`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to perform this operation.

zetSysmanMemoryGet

__ze_api_export *ze_result_t* __zecall **zetSysmanMemoryGet**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_mem_handle_t **phMemory*)

Get handle of memory modules.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phMemory`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanMemoryGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanMemoryGetProperties**(zet_sysman_mem_handle_t *hMemory*, *zet_mem_properties_t* **pProperties*)

Get memory properties.

Parameters

- `hMemory`: Handle for the component.
- `pProperties`: Will contain memory properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMemory`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanMemoryGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanMemoryGetState**(zet_sysman_mem_handle_t *hMemory*, *zet_mem_state_t* **pState*)

Get memory state - health, allocated.

Parameters

- `hMemory`: Handle for the component.
- `pState`: Will contain the current health and allocated memory.

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMemory`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`

zetSysmanMemoryGetBandwidth

__ze_api_export *ze_result_t* __zecall **`zetSysmanMemoryGetBandwidth`**(zet_sysman_mem_handle_t *hMemory*, *zet_mem_bandwidth_t* **pBandwidth*)

Get memory bandwidth.

Parameters

- `hMemory`: Handle for the component.
- `pBandwidth`: Will contain a snapshot of the bandwidth counters.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hMemory`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pBandwidth`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to query this telemetry.

zetSysmanFabricPortGet

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_fabric_port_handle_t **phPort*)

Get handle of Fabric ports in a device.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phPort`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanFabricPortGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGetProperties** (zet_sysman_fabric_port_handle_t *hPort*, *zet_fabric_port_properties_t* **pProperties*)

Get Fabric port properties.

Parameters

- `hPort`: Handle for the component.
- `pProperties`: Will contain properties of the Fabric Port.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanFabricPortGetLinkType

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGetLinkType**(zet_sysman_fabric_port_handle_t *hPort*, ze_bool_t *verbose*, *zet_fabric_link_type_t* **pLinkType*)

Get Fabric port link type.

Parameters

- `hPort`: Handle for the component.
- `verbose`: Set to true to get a more detailed report.
- `pLinkType`: Will contain details about the link attached to the Fabric port.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pLinkType`

zetSysmanFabricPortGetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGetConfig**(zet_sysman_fabric_port_handle_t *hPort*, *zet_fabric_port_config_t* **pConfig*)

Get Fabric port configuration.

Parameters

- `hPort`: Handle for the component.
- `pConfig`: Will contain configuration of the Fabric Port.

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`

zetSysmanFabricPortSetConfig

__ze_api_export *ze_result_t* __zecall **`zetSysmanFabricPortSetConfig`**(zet_sysman_fabric_port_handle_t *hPort*, **`const`** *zet_fabric_port_config_t* **pConfig*)

Set Fabric port configuration.

Parameters

- `hPort`: Handle for the component.
- `pConfig`: Contains new configuration of the Fabric Port.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanFabricPortGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGetState**(zet_sysman_fabric_port_handle_t *hPort*, *zet_fabric_port_state_t* **pState*)

Get Fabric port state - status (green/yellow/red/black), reasons for link degradation or instability, current rx/tx speed.

Parameters

- `hPort`: Handle for the component.
- `pState`: Will contain the current state of the Fabric Port

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`

zetSysmanFabricPortGetThroughput

__ze_api_export *ze_result_t* __zecall **zetSysmanFabricPortGetThroughput**(zet_sysman_fabric_port_handle_t *hPort*, *zet_fabric_port_throughput_t* **pThroughput*)

Get Fabric port throughput.

Parameters

- `hPort`: Handle for the component.
- `pThroughput`: Will contain the Fabric port throughput counters and maximum bandwidth.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPort`

- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pThroughput`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to query this telemetry.

zetSysmanTemperatureGet

__ze_api_export *ze_result_t* __zecall **zetSysmanTemperatureGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_temp_handle_t **phTemperature*)

Get handle of temperature sensors.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phTemperature`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanTemperatureGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanTemperatureGetProperties** (zet_sysman_temp_handle_t *hTemperature*, *zet_temp_properties_t* **pProperties*)

Get temperature sensor properties.

Parameters

- `hTemperature`: Handle for the component.
- `pProperties`: Will contain the temperature sensor properties.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTemperature`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanTemperatureGetConfig

__ze_api_export *ze_result_t* __zecall **`zetSysmanTemperatureGetConfig`**(zet_sysman_temp_handle_t *hTemperature*, *zet_temp_config_t* **pConfig*)

Get temperature configuration for this sensor - which events are triggered and the trigger conditions.

Parameters

- `hTemperature`: Handle for the component.
- `pConfig`: Returns current configuration.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTemperature`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Temperature thresholds are not supported on this temperature sensor. Generally this is only supported for temperature sensor *ZET_TEMP_SENSORS_GLOBAL*
  - One or both of the thresholds is not supported - check *zet_temp_properties_t.isThreshold1Supported* and *zet_temp_properties_t.isThreshold2Supported*
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to request this feature.

zetSysmanTemperatureSetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanTemperatureSetConfig**(zet_sysman_temp_handle_t *hTemperature*, **const** *zet_temp_config_t* **pConfig*)

Set temperature configuration for this sensor - indicates which events are triggered and the trigger conditions.

Parameters

- `hTemperature`: Handle for the component.
- `pConfig`: New configuration.

- Events *ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL* will be triggered when temperature reaches the critical range. Use the function *zetSysmanEventSetConfig()* to start receiving this event.
- Events *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1* and *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD2* will be generated when temperature cross the thresholds set using this function. Use the function *zetSysmanEventSetConfig()* to start receiving these events.
- Only one running process can set the temperature configuration at a time. If another process attempts to change the configuration, the error *ZE_RESULT_ERROR_NOT_AVAILABLE* will be returned. The function *zetSysmanTemperatureGetConfig()* will return the process ID currently controlling these settings.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTemperature`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_UNSUPPORTED_FEATURE*
  - Temperature thresholds are not supported on this temperature sensor. Generally they are only supported for temperature sensor *ZET_TEMP_SENSORS_GLOBAL*
  - Enabling the critical temperature event is not supported - check *zet_temp_properties_t.isCriticalTempSupported*
  - One or both of the thresholds is not supported - check *zet_temp_properties_t.isThreshold1Supported* and *zet_temp_properties_t.isThreshold2Supported*
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to request this feature.
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - Another running process is controlling these settings.
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*

   - One or both the thresholds is above TjMax (see *zetSysmanFrequencyOcGetTjMax()*). Temperature thresholds must be below this value.

zetSysmanTemperatureGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanTemperatureGetState**(zet_sysman_temp_handle_t *hTemperature*, double **pTemperature*)

Get the temperature from a specified sensor.

Parameters

- `hTemperature`: Handle for the component.
- `pTemperature`: Will contain the temperature read from the specified sensor in degrees Celcius.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTemperature`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pTemperature`

zetSysmanPsuGet

__ze_api_export *ze_result_t* __zecall **zetSysmanPsuGet**(zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_psu_handle_t **phPsu*)

Get handle of power supplies.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phPsu`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*

- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanPsuGetProperties

__ze_api_export *ze_result_t* __zecall **`zetSysmanPsuGetProperties`**(zet_sysman_psu_handle_t *hPsu*, *zet_psu_properties_t* **pProperties*)

Get power supply properties.

Parameters

- `hPsu`: Handle for the component.
- `pProperties`: Will contain the properties of the power supply.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPsu`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanPsuGetState

__ze_api_export *ze_result_t* __zecall **`zetSysmanPsuGetState`**(zet_sysman_psu_handle_t *hPsu*, *zet_psu_state_t* **pState*)

Get current power supply state.

Parameters

- `hPsu`: Handle for the component.
- `pState`: Will contain the current state of the power supply.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hPsu`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`

zetSysmanFanGet

__ze_api_export *ze_result_t* __zecall **zetSysmanFanGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_fan_handle_t **phFan*)

Get handle of fans.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phFan`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanFanGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanFanGetProperties** (zet_sysman_fan_handle_t *hFan*, *zet_fan_properties_t* **pProperties*)

Get fan properties.

Parameters

- `hFan`: Handle for the component.

- `pProperties`: Will contain the properties of the fan.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFan`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanFanGetConfig

__ze_api_export *ze_result_t* __zecall **`zetSysmanFanGetConfig`**(zet_sysman_fan_handle_t *hFan*, *zet_fan_config_t* **pConfig*)

Get current fan configuration.

Parameters

- `hFan`: Handle for the component.
- `pConfig`: Will contain the current configuration of the fan.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFan`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`

zetSysmanFanSetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanFanSetConfig**(zet_sysman_fan_handle_t *hFan*, **const** *zet_fan_config_t* **pConfig*)

Set fan configuration.

Parameters

- `hFan`: Handle for the component.
- `pConfig`: New fan configuration.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hFan`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanFanGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanFanGetState**(zet_sysman_fan_handle_t *hFan*, *zet_fan_speed_units_t* *units*, uint32_t **pSpeed*)

Get current state of a fan - current mode and speed.

Parameters

- `hFan`: Handle for the component.
- `units`: The units in which the fan speed should be returned.
- `pSpeed`: Will contain the current speed of the fan in the units requested.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

    - `nullptr == hFan`
  - *ZE_RESULT_ERROR_INVALID_ENUMERATION*
    - units
  - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pSpeed`

zetSysmanLedGet

__ze_api_export *ze_result_t* __zecall **`zetSysmanLedGet`** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_led_handle_t **phLed*)

Get handle of LEDs.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phLed`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanLedGetProperties

__ze_api_export *ze_result_t* __zecall **`zetSysmanLedGetProperties`** (zet_sysman_led_handle_t *hLed*, *zet_led_properties_t* **pProperties*)

Get LED properties.

Parameters

- `hLed`: Handle for the component.
- `pProperties`: Will contain the properties of the LED.

- The application may call this function from simultaneous threads.

- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hLed`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanLedGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanLedGetState**(zet_sysman_led_handle_t *hLed*, *zet_led_state_t* **pState*)

Get current state of a LED - on/off, color.

Parameters

- `hLed`: Handle for the component.
- `pState`: Will contain the current state of the LED.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hLed`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`

zetSysmanLedSetState

__ze_api_export *ze_result_t* __zecall **zetSysmanLedSetState**(zet_sysman_led_handle_t *hLed*, **const** *zet_led_state_t* **pState*)

Set state of a LED - on/off, color.

Parameters

- `hLed`: Handle for the component.
- `pState`: New state of the LED.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hLed`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pState`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to make these modifications.

zetSysmanRasGet

__ze_api_export *ze_result_t* __zecall **zetSysmanRasGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_ras_handle_t **phRas*)

Get handle of all RAS error sets on a device.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phRas`: [optional][range(0, *pCount)] array of handle of components of this type

- A RAS error set is a collection of RAS error counters of a given type (correctable/uncorrectable) from hardware blocks contained within a sub-device or within the device.
- A device without sub-devices will typically return two handles, one for correctable errors sets and one for uncorrectable error sets.
- A device with sub-devices will return RAS error sets for each sub-device and possibly RAS error sets for hardware blocks outside the sub-devices.
- If the function completes successfully but pCount is set to 0, RAS features are not available/enabled on this device.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*

- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanRasGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanRasGetProperties**(zet_sysman_ras_handle_t *hRas*, *zet_ras_properties_t* **pProperties*)

Get RAS properties of a given RAS error set - this enables discovery of the type of RAS error set (correctable/uncorrectable) and if located on a sub-device.

Parameters

- `hRas`: Handle for the component.
- `pProperties`: Structure describing RAS properties

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hRas`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanRasGetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanRasGetConfig**(zet_sysman_ras_handle_t *hRas*, *zet_ras_config_t* **pConfig*)

Get RAS error thresholds that control when RAS events are generated.

Parameters

- `hRas`: Handle for the component.
- `pConfig`: Will be populed with the current RAS configuration - thresholds used to trigger events

- The driver maintains counters for all RAS error sets and error categories. Events are generated when errors occur. The configuration enables setting thresholds to limit when events are sent.
- When a particular RAS correctable error counter exceeds the configured threshold, the event *ZET_SYSMAN_EVENT_TYPE_RAS_CORRECTABLE_ERRORS* will be triggered.

- When a particular RAS uncorrectable error counter exceeds the configured threshold, the event *ZET_SYSMAN_EVENT_TYPE_RAS_UNCORRECTABLE_ERRORS* will be triggered.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hRas`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`

zetSysmanRasSetConfig

__ze_api_export *ze_result_t* __zecall **`zetSysmanRasSetConfig`**(zet_sysman_ras_handle_t *hRas*, **`const`** *zet_ras_config_t* **pConfig*)

Set RAS error thresholds that control when RAS events are generated.

Parameters

- `hRas`: Handle for the component.
- `pConfig`: Change the RAS configuration - thresholds used to trigger events

- The driver maintains counters for all RAS error sets and error categories. Events are generated when errors occur. The configuration enables setting thresholds to limit when events are sent.
- When a particular RAS correctable error counter exceeds the specified threshold, the event *ZET_SYSMAN_EVENT_TYPE_RAS_CORRECTABLE_ERRORS* will be generated.
- When a particular RAS uncorrectable error counter exceeds the specified threshold, the event *ZET_SYSMAN_EVENT_TYPE_RAS_UNCORRECTABLE_ERRORS* will be generated.
- Call *zetSysmanRasGetState()* and set the clear flag to true to restart event generation once counters have exceeded thresholds.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hRas`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*

  - `nullptr == pConfig`
- *ZE_RESULT_ERROR_NOT_AVAILABLE*
  - Another running process is controlling these settings.
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - Don't have permissions to set thresholds.

zetSysmanRasGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanRasGetState** (zet_sysman_ras_handle_t *hRas*, ze_bool_t *clear*, uint64_t **pTotalErrors*, *zet_ras_details_t* **pDetails*)

Get the current value of RAS error counters for a particular error set.

Parameters

- `hRas`: Handle for the component.
- `clear`: Set to 1 to clear the counters of this type
- `pTotalErrors`: The number total number of errors that have occurred
- `pDetails`: [optional] Breakdown of where errors have occurred

- Clearing errors will affect other threads/applications - the counter values will start from zero.
- Clearing errors requires write permissions.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hRas`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pTotalErrors`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - Don't have permissions to clear error counters.

zetSysmanEventGet

__ze_api_export *ze_result_t* __zecall **zetSysmanEventGet**(zet_sysman_handle_t *hSysman*, zet_sysman_event_handle_t **phEvent*)

Get the event handle for the specified device.

Parameters

- `hSysman`: Sysman handle for the device
- `phEvent`: The event handle for the specified device.

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phEvent`

zetSysmanEventGetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanEventGetConfig**(zet_sysman_event_handle_t *hEvent*, *zet_event_config_t* **pConfig*)

Find out which events are currently registered on the specified device event handler.

Parameters

- `hEvent`: The event handle for the device
- `pConfig`: Will contain the current event configuration (list of registered events).

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`

zetSysmanEventSetConfig

__ze_api_export *ze_result_t* __zecall **zetSysmanEventSetConfig**(zet_sysman_event_handle_t *hEvent*, **const** *zet_event_config_t* **pConfig*)

Set a new event configuration (list of registered events) on the specified device event handler.

Parameters

- `hEvent`: The event handle for the device
- `pConfig`: New event configuration (list of registered events).

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hEvent`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pConfig`

zetSysmanEventGetState

__ze_api_export *ze_result_t* __zecall **zetSysmanEventGetState**(zet_sysman_event_handle_t *hEvent*, ze_bool_t *clear*, uint32_t **pEvents*)

Get events that have been triggered for a specific device.

Parameters

- `hEvent`: The event handle for the device.
- `clear`: Indicates if the event list for this device should be cleared.
- `pEvents`: Bitfield of events *zet_sysman_event_type_t* that have been triggered by this device.

- If events have occurred on the specified device event handle, they are returned and the corresponding event status is cleared if the argument clear = true.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*

    - `nullptr == hEvent`
  - *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
    - `nullptr == pEvents`

zetSysmanEventListen

__ze_api_export *ze_result_t* __zecall **`zetSysmanEventListen`** (ze_driver_handle_t *hDriver*, uint32_t *timeout*, uint32_t *count*, zet_sysman_event_handle_t **phEvents*, uint32_t **pEvents*)

Wait for the specified list of event handles to receive any registered events.

Parameters

- `hDriver`: handle of the driver instance
- `timeout`: How long to wait in milliseconds for events to arrive. Set to ZET_EVENT_WAIT_NONE will check status and return immediately. Set to ZET_EVENT_WAIT_INFINITE to block until events arrive.
- `count`: Number of handles in phEvents
- `phEvents`: [range(0, count)] Handle of events that should be listened to
- `pEvents`: Bitfield of events *zet_sysman_event_type_t* that have been triggered by any of the supplied event handles. If timeout is not ZET_EVENT_WAIT_INFINITE and this value is *ZET_SYSMAN_EVENT_TYPE_NONE*, then a timeout has occurred.

- If previous events arrived and were not cleared using *zetSysmanEventGetState()*, this call will return immediately.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == phEvents`
  - `nullptr == pEvents`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to listen to events.
- *ZE_RESULT_ERROR_INVALID_ARGUMENT*
  - One or more of the supplied event handles are for devices that belong to a different driver handle.

zetSysmanDiagnosticsGet

__ze_api_export *ze_result_t* __zecall **zetSysmanDiagnosticsGet** (zet_sysman_handle_t *hSysman*, uint32_t **pCount*, zet_sysman_diag_handle_t **phDiagnostics*)

Get handle of diagnostics test suites.

Parameters

- `hSysman`: Sysman handle of the device.
- `pCount`: pointer to the number of components of this type. if count is zero, then the driver will update the value with the total number of components of this type. if count is non-zero, then driver will only retrieve that number of components. if count is larger than the number of components available, then the driver will update the value with the correct number of components that are returned.
- `phDiagnostics`: [optional][range(0, *pCount)] array of handle of components of this type

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hSysman`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanDiagnosticsGetProperties

__ze_api_export *ze_result_t* __zecall **zetSysmanDiagnosticsGetProperties** (zet_sysman_diag_handle_t *hDiagnostics*, *zet_diag_properties_t* **pProperties*)

Get properties of a diagnostics test suite.

Parameters

- `hDiagnostics`: Handle for the component.
- `pProperties`: Structure describing the properties of a diagnostics test suite

- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*

- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDiagnostics`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pProperties`

zetSysmanDiagnosticsGetTests

__ze_api_export *ze_result_t* __zecall **zetSysmanDiagnosticsGetTests**(zet_sysman_diag_handle_t *hDiagnostics*, uint32_t **pCount*, *zet_diag_test_t* **pTests*)

Get individual tests that can be run separately. Not all test suites permit running individual tests - check *zet_diag_properties_t.haveTests*.

Parameters

- `hDiagnostics`: Handle for the component.
- `pCount`: pointer to the number of tests. If count is zero, then the driver will update the value with the total number of tests available. If count is non-zero, then driver will only retrieve that number of tests. If count is larger than the number of tests available, then the driver will update the value with the correct number of tests available.
- `pTests`: [optional][range(0, *pCount)] Array of tests sorted by increasing value of *zet_diag_test_t.index*

- The list of available tests is returned in order of increasing test index *zet_diag_test_t.index*.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDiagnostics`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCount`

zetSysmanDiagnosticsRunTests

__ze_api_export *ze_result_t* __zecall **zetSysmanDiagnosticsRunTests**(zet_sysman_diag_handle_t *hDiagnostics*, uint32_t *start*, uint32_t *end*, *zet_diag_result_t* **pResult*)

Run a diagnostics test suite, either all tests or a subset of tests.

Parameters

- `hDiagnostics`: Handle for the component.
- `start`: The index of the first test to run. Set to ZET_DIAG_FIRST_TEST_INDEX to start from the beginning.
- `end`: The index of the last test to run. Set to ZET_DIAG_LAST_TEST_INDEX to complete all tests after the start test.
- `pResult`: The result of the diagnostics

- To run all tests in a test suite, set start = ZET_DIAG_FIRST_TEST_INDEX and end = ZET_DIAG_LAST_TEST_INDEX.
- If the test suite permits running individual tests, *zet_diag_properties_t.haveTests* will be true. In this case, the function *zetSysmanDiagnosticsGetTests()* can be called to get the list of tests and corresponding indices that can be supplied to the arguments start and end in this function.
- This function will block until the diagnostics have completed.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDiagnostics`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pResult`
- *ZE_RESULT_ERROR_INSUFFICIENT_PERMISSIONS*
  - User does not have permissions to perform diagnostics.

20.6.2 Sysman Enums

zet_sysman_version_t

enum zet_sysman_version_t

API version of Sysman.

Values:

enumerator ZET_SYSMAN_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

zet_engine_type_t

enum zet_engine_type_t

Types of accelerator engines.

Values:

enumerator ZET_ENGINE_TYPE_OTHER = 0
: Undefined types of accelerators.

enumerator ZET_ENGINE_TYPE_COMPUTE
: Engines that process compute kernels.

enumerator ZET_ENGINE_TYPE_3D
: Engines that process 3D content.

enumerator ZET_ENGINE_TYPE_MEDIA
: Engines that process media workloads.

enumerator ZET_ENGINE_TYPE_DMA
: Engines that copy blocks of data.

zet_sched_mode_t

enum zet_sched_mode_t

Scheduler mode.

Values:

enumerator ZET_SCHED_MODE_TIMEOUT = 0
: Multiple applications or contexts are submitting work to the hardware. When higher priority work arrives, the scheduler attempts to pause the current executing work within some timeout interval, then submits the other work.

enumerator ZET_SCHED_MODE_TIMESLICE
: The scheduler attempts to fairly timeslice hardware execution time between multiple contexts submitting work to the hardware concurrently.

enumerator ZET_SCHED_MODE_EXCLUSIVE
: Any application or context can run indefinitely on the hardware without being preempted or terminated. All pending work for other contexts must wait until the running context completes with no further submitted work.

enumerator ZET_SCHED_MODE_COMPUTE_UNIT_DEBUG
: Scheduler ensures that submission of workloads to the hardware is optimized for compute unit debugging.

zet_perf_profile_t

enum zet_perf_profile_t

Workload performance profiles.

Values:

enumerator ZET_PERF_PROFILE_BALANCED = 0
: The hardware is configured to strike a balance between compute and memory resources. This is the default profile when the device boots/resets.

enumerator ZET_PERF_PROFILE_COMPUTE_BOUNDED
: The hardware is configured to prioritize performance of the compute units.

enumerator ZET_PERF_PROFILE_MEMORY_BOUNDED
The hardware is configured to prioritize memory throughput.

zet_repair_status_t

enum zet_repair_status_t
Device repair status.

Values:

enumerator ZET_REPAIR_STATUS_UNSUPPORTED = 0
The device does not support in-field repairs.

enumerator ZET_REPAIR_STATUS_NOT_PERFORMED
The device has never been repaired.

enumerator ZET_REPAIR_STATUS_PERFORMED
The device has been repaired.

zet_pci_link_status_t

enum zet_pci_link_status_t
PCI link status.

Values:

enumerator ZET_PCI_LINK_STATUS_GREEN = 0
The link is up and operating as expected.

enumerator ZET_PCI_LINK_STATUS_YELLOW
The link is up but has quality and/or bandwidth degradation.

enumerator ZET_PCI_LINK_STATUS_RED
The link has stability issues and preventing workloads making forward progress

zet_pci_link_qual_issues_t

enum zet_pci_link_qual_issues_t
PCI link quality degradation reasons.

Values:

enumerator ZET_PCI_LINK_QUAL_ISSUES_NONE = 0
There are no quality issues with the link at this time.

enumerator ZET_PCI_LINK_QUAL_ISSUES_REPLAYS = ZE_BIT(0)
An significant number of replays are occurring.

enumerator ZET_PCI_LINK_QUAL_ISSUES_SPEED = ZE_BIT(1)
There is a degradation in the maximum bandwidth of the link.

zet_pci_link_stab_issues_t

enum zet_pci_link_stab_issues_t

PCI link stability issues.

Values:

enumerator ZET_PCI_LINK_STAB_ISSUES_NONE = 0

There are no connection stability issues at this time.

enumerator ZET_PCI_LINK_STAB_ISSUES_RETRAINING = ZE_BIT(0)

Link retraining has occurred to deal with quality issues.

zet_pci_bar_type_t

enum zet_pci_bar_type_t

PCI bar types.

Values:

enumerator ZET_PCI_BAR_TYPE_CONFIG = 0

PCI configuration space.

enumerator ZET_PCI_BAR_TYPE_MMIO

MMIO registers.

enumerator ZET_PCI_BAR_TYPE_VRAM

VRAM aperture.

enumerator ZET_PCI_BAR_TYPE_ROM

ROM aperture.

enumerator ZET_PCI_BAR_TYPE_VGA_IO

Legacy VGA IO ports.

enumerator ZET_PCI_BAR_TYPE_VGA_MEM

Legacy VGA memory.

enumerator ZET_PCI_BAR_TYPE_INDIRECT_IO

Indirect IO port access.

enumerator ZET_PCI_BAR_TYPE_INDIRECT_MEM

Indirect memory access.

enumerator ZET_PCI_BAR_TYPE_OTHER

Other type of PCI bar.

zet_freq_domain_t

enum zet_freq_domain_t

Frequency domains.

Values:

enumerator ZET_FREQ_DOMAIN_GPU = 0

GPU Core Domain.

enumerator ZET_FREQ_DOMAIN_MEMORY

Local Memory Domain.

zet_freq_throttle_reasons_t

enum zet_freq_throttle_reasons_t
Frequency throttle reasons.

Values:

enumerator ZET_FREQ_THROTTLE_REASONS_NONE = 0
frequency not throttled

enumerator ZET_FREQ_THROTTLE_REASONS_AVE_PWR_CAP = ZE_BIT(0)
frequency throttled due to average power excursion (PL1)

enumerator ZET_FREQ_THROTTLE_REASONS_BURST_PWR_CAP = ZE_BIT(1)
frequency throttled due to burst power excursion (PL2)

enumerator ZET_FREQ_THROTTLE_REASONS_CURRENT_LIMIT = ZE_BIT(2)
frequency throttled due to current excursion (PL4)

enumerator ZET_FREQ_THROTTLE_REASONS_THERMAL_LIMIT = ZE_BIT(3)
frequency throttled due to thermal excursion (T > TjMax)

enumerator ZET_FREQ_THROTTLE_REASONS_PSU_ALERT = ZE_BIT(4)
frequency throttled due to power supply assertion

enumerator ZET_FREQ_THROTTLE_REASONS_SW_RANGE = ZE_BIT(5)
frequency throttled due to software supplied frequency range

enumerator ZET_FREQ_THROTTLE_REASONS_HW_RANGE = ZE_BIT(6)
range when it receives clocks

frequency throttled due to a sub block that has a lower frequency

zet_oc_mode_t

enum zet_oc_mode_t
Overclocking modes.

Values:

enumerator ZET_OC_MODE_OFF = 0
Overclocking if off - hardware is running using factory default voltages/frequencies.

enumerator ZET_OC_MODE_OVERRIDE
Overclock override mode - In this mode, a fixed user-supplied voltage is applied independent of the frequency request. The maximum permitted frequency can also be increased.

enumerator ZET_OC_MODE_INTERPOLATIVE
Overclock interpolative mode - In this mode, the voltage/frequency curve can be extended with a new voltage/frequency point that will be interpolated. The existing voltage/frequency points can also be offset (up or down) by a fixed voltage.

zet_engine_group_t

enum zet_engine_group_t

Accelerator engine groups.

Values:

enumerator ZET_ENGINE_GROUP_ALL = 0

Access information about all engines combined.

enumerator ZET_ENGINE_GROUP_COMPUTE_ALL

Access information about all compute engines combined.

enumerator ZET_ENGINE_GROUP_MEDIA_ALL

Access information about all media engines combined.

enumerator ZET_ENGINE_GROUP_COPY_ALL

Access information about all copy (blitter) engines combined.

zet_standby_type_t

enum zet_standby_type_t

Standby hardware components.

Values:

enumerator ZET_STANDBY_TYPE_GLOBAL = 0

Control the overall standby policy of the device/sub-device.

zet_standby_promo_mode_t

enum zet_standby_promo_mode_t

Standby promotion modes.

Values:

enumerator ZET_STANDBY_PROMO_MODE_DEFAULT = 0

Best compromise between performance and energy savings.

enumerator ZET_STANDBY_PROMO_MODE_NEVER

The device/component will never shutdown. This can improve performance but uses more energy.

zet_mem_type_t

enum zet_mem_type_t

Memory module types.

Values:

enumerator ZET_MEM_TYPE_HBM = 0

HBM memory.

enumerator ZET_MEM_TYPE_DDR

DDR memory.

enumerator ZET_MEM_TYPE_SRAM

SRAM memory.

enumerator ZET_MEM_TYPE_L1
L1 cache.

enumerator ZET_MEM_TYPE_L3
L3 cache.

enumerator ZET_MEM_TYPE_GRF
Execution unit register file.

enumerator ZET_MEM_TYPE_SLM
Execution unit shared local memory.

zet_mem_health_t

enum zet_mem_health_t
Memory health.

Values:

enumerator ZET_MEM_HEALTH_OK = 0
All memory channels are healthy.

enumerator ZET_MEM_HEALTH_DEGRADED
Excessive correctable errors have been detected on one or more channels. Device should be reset.

enumerator ZET_MEM_HEALTH_CRITICAL
Operating with reduced memory to cover banks with too many uncorrectable errors.

enumerator ZET_MEM_HEALTH_REPLACE
Device should be replaced due to excessive uncorrectable errors.

zet_fabric_port_status_t

enum zet_fabric_port_status_t
Fabric port status.

Values:

enumerator ZET_FABRIC_PORT_STATUS_GREEN = 0
The port is up and operating as expected.

enumerator ZET_FABRIC_PORT_STATUS_YELLOW
The port is up but has quality and/or bandwidth degradation.

enumerator ZET_FABRIC_PORT_STATUS_RED
Port connection instabilities are preventing workloads making forward progress

enumerator ZET_FABRIC_PORT_STATUS_BLACK
The port is configured down.

zet_fabric_port_qual_issues_t

enum zet_fabric_port_qual_issues_t

Fabric port quality degradation reasons.

Values:

enumerator ZET_FABRIC_PORT_QUAL_ISSUES_NONE = 0
There are no quality issues with the link at this time.

enumerator ZET_FABRIC_PORT_QUAL_ISSUES_FEC = ZE_BIT(0)
Excessive FEC (forward error correction) are occurring.

enumerator ZET_FABRIC_PORT_QUAL_ISSUES_LTP_CRC = ZE_BIT(1)
Excessive LTP CRC failure induced replays are occurring.

enumerator ZET_FABRIC_PORT_QUAL_ISSUES_SPEED = ZE_BIT(2)
There is a degradation in the maximum bandwidth of the port.

zet_fabric_port_stab_issues_t

enum zet_fabric_port_stab_issues_t

Fabric port stability issues.

Values:

enumerator ZET_FABRIC_PORT_STAB_ISSUES_NONE = 0
There are no connection stability issues at this time.

enumerator ZET_FABRIC_PORT_STAB_ISSUES_TOO_MANY_REPLAYS = ZE_BIT(0)
Sequential replay failure is inducing link retraining.

enumerator ZET_FABRIC_PORT_STAB_ISSUES_NO_CONNECT = ZE_BIT(1)
A connection was never able to be established through the link.

enumerator ZET_FABRIC_PORT_STAB_ISSUES_FLAPPING = ZE_BIT(2)
The port is flapping.

zet_temp_sensors_t

enum zet_temp_sensors_t

Temperature sensors.

Values:

enumerator ZET_TEMP_SENSORS_GLOBAL = 0
The maximum temperature across all device sensors.

enumerator ZET_TEMP_SENSORS_GPU
The maximum temperature across all sensors in the GPU.

enumerator ZET_TEMP_SENSORS_MEMORY
The maximum temperature across all sensors in the local memory.

zet_psu_voltage_status_t

enum zet_psu_voltage_status_t
PSU voltage status.

Values:

enumerator ZET_PSU_VOLTAGE_STATUS_NORMAL = 0
No unusual voltages have been detected.

enumerator ZET_PSU_VOLTAGE_STATUS_OVER
Over-voltage has occurred.

enumerator ZET_PSU_VOLTAGE_STATUS_UNDER
Under-voltage has occurred.

zet_fan_speed_mode_t

enum zet_fan_speed_mode_t
Fan resource speed mode.

Values:

enumerator ZET_FAN_SPEED_MODE_DEFAULT = 0
The fan speed is operating using the hardware default settings.

enumerator ZET_FAN_SPEED_MODE_FIXED
The fan speed is currently set to a fixed value.

enumerator ZET_FAN_SPEED_MODE_TABLE
The fan speed is currently controlled dynamically by hardware based on a temp/speed table

zet_fan_speed_units_t

enum zet_fan_speed_units_t
Fan speed units.

Values:

enumerator ZET_FAN_SPEED_UNITS_RPM = 0
The fan speed is in units of revolutions per minute (rpm)

enumerator ZET_FAN_SPEED_UNITS_PERCENT
The fan speed is a percentage of the maximum speed of the fan.

zet_ras_error_type_t

enum zet_ras_error_type_t
RAS error type.

Values:

enumerator ZET_RAS_ERROR_TYPE_CORRECTABLE = 0
Errors were corrected by hardware.

enumerator ZET_RAS_ERROR_TYPE_UNCORRECTABLE
Error were not corrected.

zet_sysman_event_type_t

enum zet_sysman_event_type_t

Event types.

Values:

enumerator ZET_SYSMAN_EVENT_TYPE_NONE = 0
Specifies no events.

enumerator ZET_SYSMAN_EVENT_TYPE_DEVICE_RESET = ZE_BIT(0)
Event is triggered when the driver is going to reset the device.

enumerator ZET_SYSMAN_EVENT_TYPE_DEVICE_SLEEP_STATE_ENTER = ZE_BIT(1)
deep sleep state

Event is triggered when the driver is about to put the device into a

enumerator ZET_SYSMAN_EVENT_TYPE_DEVICE_SLEEP_STATE_EXIT = ZE_BIT(2)
sleep state

Event is triggered when the driver is waking the device up from a deep

enumerator ZET_SYSMAN_EVENT_TYPE_FREQ_THROTTLED = ZE_BIT(3)
Event is triggered when the frequency starts being throttled.

enumerator ZET_SYSMAN_EVENT_TYPE_ENERGY_THRESHOLD_CROSSED = ZE_BIT(4)
(use *zetSysmanPowerSetEnergyThreshold()* to configure).

Event is triggered when the energy consumption threshold is reached

enumerator ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL = ZE_BIT(5)
zetSysmanTemperatureSetConfig() to configure - disabled by default).

Event is triggered when the critical temperature is reached (use

enumerator ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1 = ZE_BIT(6)
zetSysmanTemperatureSetConfig() to configure - disabled by default).

Event is triggered when the temperature crosses threshold 1 (use

enumerator ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD2 = ZE_BIT(7)
zetSysmanTemperatureSetConfig() to configure - disabled by default).

Event is triggered when the temperature crosses threshold 2 (use

enumerator ZET_SYSMAN_EVENT_TYPE_MEM_HEALTH = ZE_BIT(8)
Event is triggered when the health of device memory changes.

enumerator ZET_SYSMAN_EVENT_TYPE_FABRIC_PORT_HEALTH = ZE_BIT(9)
Event is triggered when the health of fabric ports change.

enumerator ZET_SYSMAN_EVENT_TYPE_PCI_LINK_HEALTH = ZE_BIT(10)
Event is triggered when the health of the PCI link changes.

enumerator ZET_SYSMAN_EVENT_TYPE_RAS_CORRECTABLE_ERRORS = ZE_BIT(11)
Event is triggered when accelerator RAS correctable errors cross thresholds (use *zetSysmanRasSetConfig()* to configure - disabled by default).

enumerator ZET_SYSMAN_EVENT_TYPE_RAS_UNCORRECTABLE_ERRORS = ZE_BIT(12)
Event is triggered when accelerator RAS uncorrectable errors cross thresholds (use *zetSysmanRasSetConfig()* to configure - disabled by default).

enumerator ZET_SYSMAN_EVENT_TYPE_ALL = 0x0FFF
Specifies all events.

zet_diag_type_t

enum zet_diag_type_t
Diagnostic test suite type.

Values:

enumerator ZET_DIAG_TYPE_SCAN = 0
Run SCAN diagnostics.

enumerator ZET_DIAG_TYPE_ARRAY
Run Array diagnostics.

zet_diag_result_t

enum zet_diag_result_t
Diagnostic results.

Values:

enumerator ZET_DIAG_RESULT_NO_ERRORS = 0
Diagnostic completed without finding errors to repair.

enumerator ZET_DIAG_RESULT_ABORT
Diagnostic had problems running tests.

enumerator ZET_DIAG_RESULT_FAIL_CANT_REPAIR
Diagnostic had problems setting up repairs.

enumerator ZET_DIAG_RESULT_REBOOT_FOR_REPAIR
Diagnostics found errors, setup for repair and reboot is required to complete the process

20.6.3 Sysman Structures

zet_sysman_properties_t

struct zet_sysman_properties_t
Device properties.

Public Members

ze_device_properties_t **core**
[out] Core device properties

uint32_t **numSubdevices**
[out] Number of sub-devices

int8_t **serialNumber**[**ZET_STRING_PROPERTY_SIZE**]
[out] Manufacturing serial number (NULL terminated string value)

int8_t **boardNumber**[**ZET_STRING_PROPERTY_SIZE**]
[out] Manufacturing board number (NULL terminated string value)

int8_t **brandName**[**ZET_STRING_PROPERTY_SIZE**]
[out] Brand name of the device (NULL terminated string value)

int8_t **modelName**[**ZET_STRING_PROPERTY_SIZE**]
[out] Model name of the device (NULL terminated string value)

int8_t **vendorName**[**ZET_STRING_PROPERTY_SIZE**]
[out] Vendor name of the device (NULL terminated string value)

int8_t **driverVersion**[**ZET_STRING_PROPERTY_SIZE**]
[out] Installed driver version (NULL terminated string value)

zet_sched_timeout_properties_t

struct zet_sched_timeout_properties_t
Configuration for timeout scheduler mode (*ZET_SCHED_MODE_TIMEOUT*)

Public Members

uint64_t **watchdogTimeout**
[in,out] The maximum time in microseconds that the scheduler will wait for a batch of work submitted to a hardware engine to complete or to be preempted so as to run another context. If this time is exceeded, the hardware engine is reset and the context terminated. If set to ZET_SCHED_WATCHDOG_DISABLE, a running workload can run as long as it wants without being terminated, but preemption attempts to run other contexts are permitted but not enforced.

zet_sched_timeslice_properties_t

struct zet_sched_timeslice_properties_t
Configuration for timeslice scheduler mode (*ZET_SCHED_MODE_TIMESLICE*)

Public Members

uint64_t **interval**
[in,out] The average interval in microseconds that a submission for a context will run on a hardware engine before being preempted out to run a pending submission for another context.

uint64_t **yieldTimeout**
[in,out] The maximum time in microseconds that the scheduler will wait to preempt a workload running on an engine before deciding to reset the hardware engine and terminating the associated context.

zet_process_state_t

struct zet_process_state_t
Contains information about a process that has an open connection with this device.

- The application can use the process ID to query the OS for the owner and the path to the executable.

Public Members

uint32_t **processId**
[out] Host OS process ID.

int64_t **memSize**
[out] Device memory size in bytes allocated by this process (may not necessarily be resident on the device at the time of reading).

int64_t **engines**
[out] Bitfield of accelerator engines being used by this process (or 1<<*zet_engine_type_t* together).

zet_pci_address_t

struct zet_pci_address_t
PCI address.

Public Members

uint32_t **domain**
[out] BDF domain

uint32_t **bus**
[out] BDF bus

uint32_t **device**
[out] BDF device

uint32_t **function**
[out] BDF function

zet_pci_speed_t

struct zet_pci_speed_t
PCI speed.

Public Members

uint32_t **gen**
[out] The link generation

uint32_t **width**
[out] The number of lanes

uint64_t **maxBandwidth**
[out] The maximum bandwidth in bytes/sec

zet_pci_properties_t

struct zet_pci_properties_t
Static PCI properties.

Public Members

zet_pci_address_t **address**
[out] The BDF address

zet_pci_speed_t **maxSpeed**
[out] Fastest port configuration supported by the device.

zet_pci_state_t

struct zet_pci_state_t
Dynamic PCI state.

Public Members

zet_pci_link_status_t **status**
[out] The current status of the port

zet_pci_link_qual_issues_t **qualityIssues**
[out] If status is *ZET_PCI_LINK_STATUS_YELLOW*, this gives a bitfield of quality issues that have been detected

zet_pci_link_stab_issues_t **stabilityIssues**
[out] If status is *ZET_PCI_LINK_STATUS_RED*, this gives a bitfield of reasons for the connection instability

zet_pci_speed_t **speed**
[out] The current port configure speed

zet_pci_bar_properties_t

struct zet_pci_bar_properties_t
Properties of a pci bar.

Public Members

zet_pci_bar_type_t **type**
[out] The type of bar

uint32_t **index**
[out] The index of the bar

uint64_t **base**
[out] Base address of the bar.

uint64_t **size**
[out] Size of the bar.

zet_pci_stats_t

struct zet_pci_stats_t

PCI stats counters.

- Percent throughput is calculated by taking two snapshots (s1, s2) and using the equation: bw = 10^6 * ((s2.rxCounter - s1.rxCounter) + (s2.txCounter - s1.txCounter)) / (s2.maxBandwidth * (s2.timestamp - s1.timestamp))
- Percent replays is calculated by taking two snapshots (s1, s2) and using the equation: replay = 10^6 * (s2.replayCounter - s1.replayCounter) / (s2.maxBandwidth * (s2.timestamp - s1.timestamp))

Public Members

uint64_t **timestamp**

[out] Monotonic timestamp counter in microseconds when the measurement was made. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

uint64_t **replayCounter**

[out] Monotonic counter for the number of replay packets

uint64_t **packetCounter**

[out] Monotonic counter for the number of packets

uint64_t **rxCounter**

[out] Monotonic counter for the number of bytes received

uint64_t **txCounter**

[out] Monotonic counter for the number of bytes transmitted (including replays)

uint64_t **maxBandwidth**

[out] The maximum bandwidth in bytes/sec under the current configuration

zet_power_properties_t

struct zet_power_properties_t

Properties related to device power settings.

Public Members

ze_bool_t **onSubdevice**

[out] True if this resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**

[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **canControl**

[out] Software can change the power limits of this domain assuming the user has permissions.

ze_bool_t **isEnergyThresholdSupported**

[out] Indicates if this power domain supports the energy threshold event (*ZET_SYSMAN_EVENT_TYPE_ENERGY_THRESHOLD_CROSSED*).

uint32_t **maxLimit**
: [out] The maximum power limit in milliwatts that can be requested.

zet_power_energy_counter_t

struct zet_power_energy_counter_t

Energy counter snapshot.

- Average power is calculated by taking two snapshots (s1, s2) and using the equation: PowerWatts = (s2.energy - s1.energy) / (s2.timestamp - s1.timestamp)

Public Members

uint64_t **energy**
: [out] The monotonic energy counter in microjoules.

uint64_t **timestamp**
: [out] Microsecond timestamp when energy was captured. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

zet_power_sustained_limit_t

struct zet_power_sustained_limit_t

Sustained power limits.

- The power controller (Punit) will throttle the operating frequency if the power averaged over a window (typically seconds) exceeds this limit.

Public Members

ze_bool_t **enabled**
: [in,out] indicates if the limit is enabled (true) or ignored (false)

uint32_t **power**
: [in,out] power limit in milliwatts

uint32_t **interval**
: [in,out] power averaging window (Tau) in milliseconds

zet_power_burst_limit_t

struct zet_power_burst_limit_t

Burst power limit.

- The power controller (Punit) will throttle the operating frequency of the device if the power averaged over a few milliseconds exceeds a limit known as PL2. Typically PL2 > PL1 so that it permits the frequency to burst higher for short periods than would be otherwise permitted by PL1.

Public Members

ze_bool_t **enabled**
[in,out] indicates if the limit is enabled (true) or ignored (false)

uint32_t **power**
[in,out] power limit in milliwatts

zet_power_peak_limit_t

struct zet_power_peak_limit_t
Peak power limit.

- The power controller (Punit) will preemptively throttle the operating frequency of the device when the instantaneous power exceeds this limit. The limit is known as PL4. It expresses the maximum power that can be drawn from the power supply.
- If this power limit is removed or set too high, the power supply will generate an interrupt when it detects an overcurrent condition and the power controller will throttle the device frequencies down to min. It is thus better to tune the PL4 value in order to avoid such excursions.

Public Members

uint32_t **powerAC**
[in,out] power limit in milliwatts for the AC power source.

uint32_t **powerDC**
[in,out] power limit in milliwatts for the DC power source. This is ignored if the product does not have a battery.

zet_energy_threshold_t

struct zet_energy_threshold_t
Energy threshold.

- .

Public Members

ze_bool_t **enable**
[in,out] Indicates if the energy threshold is enabled.

double **threshold**
[in,out] The energy threshold in Joules. Will be 0.0 if no threshold has been set.

uint32_t **processId**
[in,out] The host process ID that set the energy threshold. Will be 0xFFFFFFFF if no threshold has been set.

zet_freq_properties_t

struct zet_freq_properties_t

Frequency properties.

- Indicates if this frequency domain can be overclocked (if true, functions such as *zetSysmanFrequencyOcSetConfig()* are supported).
- The min/max hardware frequencies are specified for non-overclock configurations. For overclock configurations, use *zetSysmanFrequencyOcGetConfig()* to determine the maximum frequency that can be requested.
- If step is non-zero, the available frequencies are (min, min + step, min + 2xstep, . . . , max). Otherwise, call *zetSysmanFrequencyGetAvailableClocks()* to get the list of frequencies that can be requested.

Public Members

zet_freq_domain_t **type**
[out] The hardware block that this frequency domain controls (GPU, memory, . . .)

ze_bool_t **onSubdevice**
[out] True if this resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **canControl**
[out] Indicates if software can control the frequency of this domain assuming the user has permissions

ze_bool_t **isThrottleEventSupported**
[out] Indicates if software can register to receive event *ZET_SYSMAN_EVENT_TYPE_FREQ_THROTTLED*

double **min**
[out] The minimum hardware clock frequency in units of MHz

double **max**
[out] The maximum non-overclock hardware clock frequency in units of MHz.

double **step**
[out] The minimum step-size for clock frequencies in units of MHz. The hardware will clamp intermediate frequencies to lowest multiplier of this number.

zet_freq_range_t

struct zet_freq_range_t

Frequency range between which the hardware can operate.

Public Members

double **min**
: [in,out] The min frequency in MHz below which hardware frequency management will not request frequencies. Setting to 0 will use the hardware default value.

double **max**
: [in,out] The max frequency in MHz above which hardware frequency management will not request frequencies. Setting to 0 will use the hardware default value.

zet_freq_state_t

struct zet_freq_state_t
Frequency state.

Public Members

double **request**
: [out] The current frequency request in MHz.

double **tdp**
: [out] The maximum frequency in MHz supported under the current TDP conditions

double **efficient**
: [out] The efficient minimum frequency in MHz

double **actual**
: [out] The resolved frequency in MHz

uint32_t **throttleReasons**
: [out] The reasons that the frequency is being limited by the hardware (Bitfield of *zet_freq_throttle_reasons_t*).

zet_freq_throttle_time_t

struct zet_freq_throttle_time_t
Frequency throttle time snapshot.

- Percent time throttled is calculated by taking two snapshots (s1, s2) and using the equation: throttled = (s2.throttleTime - s1.throttleTime) / (s2.timestamp - s1.timestamp)

Public Members

uint64_t **throttleTime**
: [out] The monotonic counter of time in microseconds that the frequency has been limited by the hardware.

uint64_t **timestamp**
: [out] Microsecond timestamp when throttleTime was captured. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

zet_oc_capabilities_t

struct zet_oc_capabilities_t

Overclocking properties.

- Provides all the overclocking capabilities and properties supported by the device for the frequency domain.

Public Members

ze_bool_t **isOcSupported**
[out] Indicates if any overclocking features are supported on this frequency domain.

double **maxFactoryDefaultFrequency**
[out] Factory default non-overclock maximum frequency in Mhz.

double **maxFactoryDefaultVoltage**
[out] Factory default voltage used for the non-overclock maximum frequency in MHz.

double **maxOcFrequency**
[out] Maximum hardware overclocking frequency limit in Mhz.

double **minOcVoltageOffset**
[out] The minimum voltage offset that can be applied to the voltage/frequency curve. Note that this number can be negative.

double **maxOcVoltageOffset**
[out] The maximum voltage offset that can be applied to the voltage/frequency curve.

double **maxOcVoltage**
[out] The maximum overclock voltage that hardware supports.

ze_bool_t **isTjMaxSupported**
[out] Indicates if the maximum temperature limit (TjMax) can be changed for this frequency domain.

ze_bool_t **isIccMaxSupported**
[out] Indicates if the maximum current (IccMax) can be changed for this frequency domain.

ze_bool_t **isHighVoltModeCapable**
[out] Indicates if this frequency domains supports a feature to set very high voltages.

ze_bool_t **isHighVoltModeEnabled**
[out] Indicates if very high voltages are permitted on this frequency domain.

zet_oc_config_t

struct zet_oc_config_t

Overclocking configuration.

- Overclock settings

Public Members

zet_oc_mode_t **mode**
[in,out] Overclock Mode *zet_oc_mode_t*.

double **frequency**
[in,out] Overclocking Frequency in MHz. This cannot be greater than *zet_oc_capabilities_t.maxOcFrequency*.

double **voltageTarget**
[in,out] Overclock voltage in Volts. This cannot be greater than *zet_oc_capabilities_t.maxOcVoltage*.

double **voltageOffset**
[in,out] This voltage offset is applied to all points on the voltage/frequency curve, include the new overclock voltageTarget. It can be in the range (*zet_oc_capabilities_t.minOcVoltageOffset*, *zet_oc_capabilities_t.maxOcVoltageOffset*).

zet_engine_properties_t

struct zet_engine_properties_t
Engine group properties.

Public Members

zet_engine_group_t **type**
[out] The engine group

ze_bool_t **onSubdevice**
[out] True if this resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

zet_engine_stats_t

struct zet_engine_stats_t
Engine activity counters.

- Percent utilization is calculated by taking two snapshots (s1, s2) and using the equation: util = (s2.activeTime - s1.activeTime) / (s2.timestamp - s1.timestamp)

Public Members

uint64_t **activeTime**
[out] Monotonic counter for time in microseconds that this resource is actively running workloads.

uint64_t **timestamp**
[out] Monotonic timestamp counter in microseconds when activeTime counter was sampled. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

zet_standby_properties_t

struct zet_standby_properties_t
Standby hardware component properties.

Public Members

zet_standby_type_t **type**
[out] Which standby hardware component this controls

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

zet_firmware_properties_t

struct zet_firmware_properties_t
Firmware properties.

Public Members

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **canControl**
[out] Indicates if software can flash the firmware assuming the user has permissions

int8_t **name**[**ZET_STRING_PROPERTY_SIZE**]
[out] NULL terminated string value

int8_t **version**[**ZET_STRING_PROPERTY_SIZE**]
[out] NULL terminated string value

zet_mem_properties_t

struct zet_mem_properties_t
Memory properties.

Public Members

zet_mem_type_t **type**
[out] The memory type

ze_bool_t **onSubdevice**
[out] True if this resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

uint64_t **physicalSize**
[out] Physical memory size in bytes

zet_mem_state_t

struct zet_mem_state_t
Memory state - health, allocated.

- Percent allocation is given by 100 * allocatedSize / maxSize.
- Percent free is given by 100 * (maxSize - allocatedSize) / maxSize.

Public Members

zet_mem_health_t **health**
[out] Indicates the health of the memory

uint64_t **allocatedSize**
[out] The total allocated bytes

uint64_t **maxSize**
[out] The total allocatable memory in bytes (can be less than *zet_mem_properties_t.physicalSize*)

zet_mem_bandwidth_t

struct zet_mem_bandwidth_t
Memory bandwidth.

- Percent bandwidth is calculated by taking two snapshots (s1, s2) and using the equation: bw = 10^6 * ((s2.readCounter - s1.readCounter) + (s2.writeCounter - s1.writeCounter)) / (s2.maxBandwidth * (s2.timestamp - s1.timestamp))

Public Members

uint64_t **readCounter**
[out] Total bytes read from memory

uint64_t **writeCounter**
[out] Total bytes written to memory

uint64_t **maxBandwidth**
[out] Current maximum bandwidth in units of bytes/sec

uint64_t **timestamp**
[out] The timestamp when these measurements were sampled. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

zet_fabric_port_uuid_t

struct zet_fabric_port_uuid_t
Fabric port universal unique id (UUID)

Public Members

uint8_t **id**[**ZET_MAX_FABRIC_PORT_UUID_SIZE**]
[out] Frabric port universal unique id

zet_fabric_port_speed_t

struct zet_fabric_port_speed_t
Fabric port speed in one direction.

Public Members

uint64_t **bitRate**
[out] Bits/sec that the link is operating at

uint32_t **width**
[out] The number of lanes

uint64_t **maxBandwidth**
[out] The maximum bandwidth in bytes/sec

zet_fabric_port_properties_t

struct zet_fabric_port_properties_t
Fabric port properties.

Public Members

int8_t **model**[**ZET_MAX_FABRIC_PORT_MODEL_SIZE**]
[out] Description of port technology

ze_bool_t **onSubdevice**
[out] True if the port is located on a sub-device; false means that the port is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

zet_fabric_port_uuid_t **portUuid**
[out] The port universal unique id

zet_fabric_port_speed_t **maxRxSpeed**
[out] Maximum bandwidth supported by the receive side of the port

zet_fabric_port_speed_t **maxTxSpeed**
[out] Maximum bandwidth supported by the transmit side of the port

zet_fabric_link_type_t

struct zet_fabric_link_type_t
Provides information about the fabric link attached to a port.

Public Members

int8_t **desc**[**ZET_MAX_FABRIC_LINK_TYPE_SIZE**]
[out] This provides a textural description of a link attached to a port. It contains the following information:

- Link material
- Link technology
- Cable manufacturer
- Temperature
- Power
- Attachment type:
  - Disconnected
  - Hardwired/fixed/etched connector
  - Active copper
  - QSOP
  - AOC

zet_fabric_port_config_t

struct zet_fabric_port_config_t
Fabric port configuration.

Public Members

ze_bool_t **enabled**
[in,out] Port is configured up/down

ze_bool_t **beaconing**
[in,out] Beaconing is configured on/off

zet_fabric_port_state_t

struct zet_fabric_port_state_t

Fabric port state.

Public Members

zet_fabric_port_status_t **status**

[out] The current status of the port

zet_fabric_port_qual_issues_t **qualityIssues**

[out] If status is *ZET_FABRIC_PORT_STATUS_YELLOW*, this gives a bitfield of quality issues that have been detected

zet_fabric_port_stab_issues_t **stabilityIssues**

[out] If status is *ZET_FABRIC_PORT_STATUS_RED*, this gives a bitfield of reasons for the connection instability

zet_fabric_port_speed_t **rxSpeed**

[out] Current maximum receive speed

zet_fabric_port_speed_t **txSpeed**

[out] Current maximum transmit speed

zet_fabric_port_throughput_t

struct zet_fabric_port_throughput_t

Fabric port throughput.

- Percent throughput is calculated by taking two snapshots (s1, s2) and using the equation:
- rx_bandwidth = 10^6 * (s2.rxCounter - s1.rxCounter) / (s2.rxMaxBandwidth * (s2.timestamp - s1.timestamp))
- tx_bandwidth = 10^6 * (s2.txCounter - s1.txCounter) / (s2.txMaxBandwidth * (s2.timestamp - s1.timestamp))

Public Members

uint64_t **timestamp**

[out] Monotonic timestamp counter in microseconds when the measurement was made. No assumption should be made about the absolute value of the timestamp. It should only be used to calculate delta time between two snapshots of the same structure. Never take the delta of this timestamp with the timestamp from a different structure.

uint64_t **rxCounter**

[out] Monotonic counter for the number of bytes received

uint64_t **txCounter**

[out] Monotonic counter for the number of bytes transmitted

uint64_t **rxMaxBandwidth**

[out] The current maximum bandwidth in bytes/sec for receiving packats

uint64_t **txMaxBandwidth**

[out] The current maximum bandwidth in bytes/sec for transmitting packets

zet_temp_properties_t

struct zet_temp_properties_t

Temperature sensor properties.

Public Members

zet_temp_sensors_t **type**

[out] Which part of the device the temperature sensor measures

ze_bool_t **onSubdevice**

[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**

[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **isCriticalTempSupported**

[out] Indicates if the critical temperature event *ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL* is supported

ze_bool_t **isThreshold1Supported**

[out] Indicates if the temperature threshold 1 event *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1* is supported

ze_bool_t **isThreshold2Supported**

[out] Indicates if the temperature threshold 2 event *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD2* is supported

zet_temp_threshold_t

struct zet_temp_threshold_t

Temperature sensor threshold.

Public Members

ze_bool_t **enableLowToHigh**

[in,out] Trigger an event when the temperature crosses from below the threshold to above.

ze_bool_t **enableHighToLow**

[in,out] Trigger an event when the temperature crosses from above the threshold to below.

double **threshold**

[in,out] The threshold in degrees Celcius.

zet_temp_config_t

struct zet_temp_config_t
Temperature configuration - which events should be triggered and the trigger conditions.

Public Members

ze_bool_t **enableCritical**
[in,out] Indicates if event *ZET_SYSMAN_EVENT_TYPE_TEMP_CRITICAL* should be triggered by the driver.

zet_temp_threshold_t **threshold1**
[in,out] Configuration controlling if and when event *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD1* should be triggered by the driver.

zet_temp_threshold_t **threshold2**
[in,out] Configuration controlling if and when event *ZET_SYSMAN_EVENT_TYPE_TEMP_THRESHOLD2* should be triggered by the driver.

uint32_t **processId**
[out] Host processId that set this configuration (ignored when setting the configuration).

zet_psu_properties_t

struct zet_psu_properties_t
Static properties of the power supply.

Public Members

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **haveFan**
[out] True if the power supply has a fan

uint32_t **ampLimit**
[out] The maximum electrical current in amperes that can be drawn

zet_psu_state_t

struct zet_psu_state_t
Dynamic state of the power supply.

Public Members

zet_psu_voltage_status_t **`voltStatus`**
[out] The current PSU voltage status

ze_bool_t **`fanFailed`**
[out] Indicates if the fan has failed

uint32_t **`temperature`**
[out] Read the current heatsink temperature in degrees Celsius.

uint32_t **`current`**
[out] The amps being drawn in amperes

zet_fan_temp_speed_t

`struct zet_fan_temp_speed_t`
Fan temperature/speed pair.

Public Members

uint32_t **`temperature`**
[in,out] Temperature in degrees Celsius.

uint32_t **`speed`**
[in,out] The speed of the fan

zet_fan_speed_units_t **`units`**
[in,out] The units of the member speed

zet_fan_properties_t

`struct zet_fan_properties_t`
Fan properties.

Public Members

ze_bool_t **`onSubdevice`**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **`subdeviceId`**
[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **`canControl`**
[out] Indicates if software can control the fan speed assuming the user has permissions

uint32_t **`maxSpeed`**
[out] The maximum RPM of the fan

uint32_t **`maxPoints`**
[out] The maximum number of points in the fan temp/speed table

zet_fan_config_t

struct zet_fan_config_t
Fan configuration.

Public Members

zet_fan_speed_mode_t **mode**
[in,out] The fan speed mode (fixed, temp-speed table)

uint32_t **speed**
[in,out] The fixed fan speed setting

zet_fan_speed_units_t **speedUnits**
[in,out] The units of the fixed fan speed setting

uint32_t **numPoints**
[in,out] The number of valid points in the fan speed table

zet_fan_temp_speed_t **table**[**ZET_FAN_TEMP_SPEED_PAIR_COUNT**]
[in,out] Array of temperature/fan speed pairs

zet_led_properties_t

struct zet_led_properties_t
LED properties.

Public Members

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

ze_bool_t **canControl**
[out] Indicates if software can control the LED assuming the user has permissions

ze_bool_t **haveRGB**
[out] Indicates if the LED is RGB capable

zet_led_state_t

struct zet_led_state_t
LED state.

Public Members

ze_bool_t **isOn**
[in,out] Indicates if the LED is on or off

uint8_t **red**
[in,out][range(0, 255)] The LED red value

uint8_t **green**
[in,out][range(0, 255)] The LED green value

uint8_t **blue**
[in,out][range(0, 255)] The LED blue value

zet_ras_properties_t

struct zet_ras_properties_t
RAS properties.

Public Members

zet_ras_error_type_t **type**
[out] The type of RAS error

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

zet_ras_details_t

struct zet_ras_details_t
RAS error details.

Public Members

uint64_t **numResets**
[out] The number of device resets that have taken place

uint64_t **numProgrammingErrors**
[out] The number of hardware exceptions generated by the way workloads have programmed the hardware

uint64_t **numDriverErrors**
[out] The number of low level driver communication errors have occurred

uint64_t **numComputeErrors**
[out] The number of errors that have occurred in the compute accelerator hardware

uint64_t **numNonComputeErrors**
[out] The number of errors that have occurred in the fixed-function accelerator hardware

uint64_t **numCacheErrors**
[out] The number of errors that have occurred in caches (L1/L3/register file/shared local memory/sampler)

uint64_t **numDisplayErrors**
[out] The number of errors that have occurred in the display

zet_ras_config_t

struct zet_ras_config_t
RAS error configuration - thresholds used for triggering RAS events (*ZET_SYSMAN_EVENT_TYPE_RAS_CORRECTABLE_ERRORS*, *ZET_SYSMAN_EVENT_TYPE_RAS_UNCORRECTABLE_ER*

- The driver maintains a total counter which is updated every time a hardware block covered by the corresponding RAS error set notifies that an error has occurred. When this total count goes above the totalThreshold specified below, a RAS event is triggered.
- The driver also maintains a counter for each category of RAS error (see *zet_ras_details_t* for a breakdown). Each time a hardware block of that category notifies that an error has occurred, that corresponding category counter is updated. When it goes above the threshold specified in detailedThresholds, a RAS event is triggered.

Public Members

uint64_t **totalThreshold**
[in,out] If the total RAS errors exceeds this threshold, the event will be triggered. A value of 0ULL disables triggering the event based on the total counter.

zet_ras_details_t **detailedThresholds**
[in,out] If the RAS errors for each category exceed the threshold for that category, the event will be triggered. A value of 0ULL will disable an event being triggered for that category.

uint32_t **processId**
[out] Host processId that set this configuration (ignored when setting the configuration).

zet_event_config_t

struct zet_event_config_t
Event configuration for a device.

Public Members

uint32_t **registered**
[in,out] List of registered events (Bitfield of events *zet_sysman_event_type_t*). *ZET_SYSMAN_EVENT_TYPE_NONE* indicates there are no registered events. *ZET_SYSMAN_EVENT_TYPE_ALL* indicates that all events are registered.

zet_diag_test_t

struct zet_diag_test_t
Diagnostic test.

Public Members

uint32_t **index**
[out] Index of the test

char **name[ZET_STRING_PROPERTY_SIZE]**
[out] Name of the test

zet_diag_properties_t

struct zet_diag_properties_t
Diagnostics test suite properties.

Public Members

zet_diag_type_t **type**
[out] The type of diagnostics test suite

ze_bool_t **onSubdevice**
[out] True if the resource is located on a sub-device; false means that the resource is on the device of the calling Sysman handle

uint32_t **subdeviceId**
[out] If onSubdevice is true, this gives the ID of the sub-device

char **name[ZET_STRING_PROPERTY_SIZE]**
[out] Name of the diagnostics test suite

ze_bool_t **haveTests**
[out] Indicates if this test suite has individual tests which can be run separately (use the function $Sysman-DiagnosticsGetTests() to get the list of these tests)

20.7 Tracing

- Functions
  - *zetTracerCreate*
  - *zetTracerDestroy*
  - *zetTracerSetPrologues*
  - *zetTracerSetEpilogues*
  - *zetTracerSetEnabled*
- Enumerations
  - *zet_tracer_desc_version_t*
- Structures

  - *zet_tracer_desc_t*

20.7.1 Tracing Functions

zetTracerCreate

__ze_api_export *ze_result_t* __zecall **`zetTracerCreate`** (zet_driver_handle_t *hDriver*, **`const`** *zet_tracer_desc_t* **desc*, zet_tracer_handle_t **phTracer*)

Creates a tracer for the specified driver.

Parameters

- `hDriver`: handle of the driver
- `desc`: pointer to tracer descriptor
- `phTracer`: pointer to handle of tracer object created

- The tracer can only be used on the driver on which it was created.
- The tracer is created in the disabled state.
- The application may call this function from simultaneous threads.
- The implementation of this function should be lock-free.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hDriver`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == desc`
  - `nullptr == desc->pUserData`
  - `nullptr == phTracer`
- *ZE_RESULT_ERROR_UNSUPPORTED_VERSION*
  - `ZET_TRACER_DESC_VERSION_CURRENT < desc->version`
- *ZE_RESULT_ERROR_INVALID_ENUMERATION*
- *ZE_RESULT_ERROR_OUT_OF_HOST_MEMORY*

zetTracerDestroy

__ze_api_export *ze_result_t* __zecall **zetTracerDestroy**(zet_tracer_handle_t *hTracer*)

Destroys a tracer.

Parameters

- `hTracer`: [release] handle of tracer object to destroy

- The application may **not** call this function from simultaneous threads with the same tracer handle.
- The implementation of this function will stall and wait on any outstanding threads executing callbacks before freeing any Host allocations associated with this tracer.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTracer`
- *ZE_RESULT_ERROR_HANDLE_OBJECT_IN_USE*

zetTracerSetPrologues

__ze_api_export *ze_result_t* __zecall **zetTracerSetPrologues**(zet_tracer_handle_t *hTracer*, zet_core_callbacks_t **pCoreCbs*)

Sets the collection of callbacks to be executed **before** driver execution.

Parameters

- `hTracer`: handle of the tracer
- `pCoreCbs`: pointer to table of 'core' callback function pointers

- The application only needs to set the function pointers it is interested in receiving; all others should be 'nullptr'
- The application must ensure that no other threads are executing functions for which the tracing functions are changing.
- The application may **not** call this function from simultaneous threads with the same tracer handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTracer`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCoreCbs`

zetTracerSetEpilogues

__ze_api_export *ze_result_t* __zecall **zetTracerSetEpilogues**(zet_tracer_handle_t *hTracer*, zet_core_callbacks_t **pCoreCbs*)

Sets the collection of callbacks to be executed **after** driver execution.

Parameters

- `hTracer`: handle of the tracer
- `pCoreCbs`: pointer to table of 'core' callback function pointers

- The application only needs to set the function pointers it is interested in receiving; all others should be 'nullptr'
- The application must ensure that no other threads are executing functions for which the tracing functions are changing.
- The application may **not** call this function from simultaneous threads with the same tracer handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTracer`
- *ZE_RESULT_ERROR_INVALID_NULL_POINTER*
  - `nullptr == pCoreCbs`

zetTracerSetEnabled

__ze_api_export *ze_result_t* __zecall **zetTracerSetEnabled**(zet_tracer_handle_t *hTracer*, ze_bool_t *enable*)

Enables (or disables) the tracer.

Parameters

- `hTracer`: handle of the tracer
- `enable`: enable the tracer if true; disable if false

- The application may **not** call this function from simultaneous threads with the same tracer handle.

Return

- *ZE_RESULT_SUCCESS*
- *ZE_RESULT_ERROR_UNINITIALIZED*
- *ZE_RESULT_ERROR_DEVICE_LOST*
- *ZE_RESULT_ERROR_INVALID_NULL_HANDLE*
  - `nullptr == hTracer`

20.7.2 Tracing Enums

zet_tracer_desc_version_t

enum zet_tracer_desc_version_t

API version of *zet_tracer_desc_t*.

Values:

enumerator ZET_TRACER_DESC_VERSION_CURRENT = ZE_MAKE_VERSION(0, 91)

version 0.91

20.7.3 Tracing Structures

zet_tracer_desc_t

struct zet_tracer_desc_t

Tracer descriptor.

Public Members

zet_tracer_desc_version_t **version**

[in] *ZET_TRACER_DESC_VERSION_CURRENT*

void ***pUserData**

[in] pointer passed to every tracer's callbacks

CHAPTER

TWENTYONE

LEGAL NOTICES AND DISCLAIMERS

INDEX