INTEL® PERCEPTUAL COMPUTING SDK

Reference Manual

Module Implementation

API Version 1.0

LEGAL DISCLAIMER

THIS DOCUMENT CONTAINS INFORMATION ON PRODUCTS IN THE DESIGN PHASE OF DEVELOPMENT.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

INTEL MAY MAKE CHANGES TO SPECIFICATIONS AND PRODUCT DESCRIPTIONS AT ANY TIME, WITHOUT NOTICE. DESIGNERS MUST NOT RELY ON THE ABSENCE OR CHARACTERISTICS OF ANY FEATURES OR INSTRUCTIONS MARKED "RESERVED" OR "UNDEFINED." INTEL RESERVES THESE FOR FUTURE DEFINITION AND SHALL HAVE NO RESPONSIBILITY WHATSOEVER FOR CONFLICTS OR INCOMPATIBILITIES ARISING FROM FUTURE CHANGES TO THEM. THE INFORMATION HERE IS SUBJECT TO CHANGE WITHOUT NOTICE. DO NOT FINALIZE A DESIGN WITH THIS INFORMATION.

THE PRODUCTS DESCRIBED IN THIS DOCUMENT MAY CONTAIN DESIGN DEFECTS OR ERRORS KNOWN AS ERRATA WHICH MAY CAUSE THE PRODUCT TO DEVIATE FROM PUBLISHED SPECIFICATIONS. CURRENT CHARACTERIZED ERRATA ARE AVAILABLE ON REQUEST.

CONTACT YOUR LOCAL INTEL SALES OFFICE OR YOUR DISTRIBUTOR TO OBTAIN THE LATEST SPECIFICATIONS AND BEFORE PLACING YOUR PRODUCT ORDER.

COPIES OF DOCUMENTS WHICH HAVE AN ORDER NUMBER AND ARE REFERENCED IN THIS DOCUMENT, OR OTHER INTEL LITERATURE, MAY BE OBTAINED BY CALLING 1-800-548-4725, OR BY VISITING INTEL'S WEB SITE HTTP://WWW.INTEL.COM.

ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE ANY SOFTWARE SOURCE CODE REPRINTED IN THIS DOCUMENT IS FURNISHED UNDER A SOFTWARE LICENSE AND MAY ONLY BE USED OR COPIED IN ACCORDANCE WITH THE TERMS OF THAT LICENSE

INTEL, THE INTEL LOGO, INTEL CORE, INTEL MEDIA SOFTWARE DEVELOPMENT KIT (INTEL MEDIA SDK) ARE TRADEMARKS OR REGISTERED TRADEMARKS OF INTEL CORPORATION OR ITS SUBSIDIARIES IN THE UNITED STATES AND OTHER COUNTRIES.

MPEG IS AN INTERNATIONAL STANDARD FOR VIDEO COMPRESSION/DECOMPRESSION PROMOTED BY ISO. IMPLEMENTATIONS OF MPEG CODECS, OR MPEG ENABLED PLATFORMS MAY REQUIRE LICENSES FROM VARIOUS ENTITIES, INCLUDING INTEL CORPORATION.

*OTHER NAMES AND BRANDS MAY BE CLAIMED AS THE PROPERTY OF OTHERS.

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110307

Table of Contents

Intel® Perceptual Computing SDK

The Intel® Perceptual Computing SDK is a library of pattern detection and recognition algorithm implementations exposed through standardized interfaces. The library's purpose is to lower barriers of using these algorithms and shift the application developers' focus from coding the algorithm details to innovating on the usage of these algorithms for next generation human computer experience.

This document describes the details of how to expose an interface and implement the interface under the SDK framework. The other reference manuals in the SDK describe additional algorithms and their API definitions.

Document Conventions

The SDK API uses the Verdana typeface for normal prose. With the exception of section headings and the table of contents, all code-related items appear in the `Courier New` typeface (`pxcStatus`). Hyperlinks appear in underlined boldface, such as **`pxcStatus`**.

Acronyms and Abbreviations

API	Application Programming Interface
SDK	Software Development Kit
SP	Synchronization Point

Programming Guide

The PXCMath Example

This example adds a math interface to the SDK framework. For simplicity, there is only one function in the **PXCMath** interface, an integer addition function.

Example 1 shows the interface declaration. The following restrictions apply:

- The **PXCMath** interface must derive from the **PXCBase** interface (except any callback interface, which must derive from the **PXCCallbackBase** interface).
- The interface must have a unique identifier, used by the **DynamicCast** function. Use the **PXC_CUID_OVERWRITE** macro to define a unique identifier of the interface.
- The interface contains only pure virtual functions. No constructor or destructor.
- The interface member functions must use the **PXCAPI** calling convention.
- Do not use dynamic_cast, or exception handling.

See Appendix A for details of SDK API guidelines.

```
class PXCMath:public PXCBase {
public:
       PXC_CUID_OVERWRITE(0x12345678);
       PXC_DEFINE_CONST(IMPL_SUBGROUP_MATH, 0x40000000);
       virtual pxcStatus PXCAPI Add(int i1, int i2, int *o)=0;
};
```

Example 1: The PXCMath Interface

In Example 2, the module developer implements the **PXCMath** interface. The **Math** class must derive from the **PXCBaseImpl<PXCMath>** interface, which is a template class that provides the default implementation of the **PXCBase** interface. There are no other restrictions to the **Math** class implementation. For example, the **Math** class can use constructor, destructor, private, or static data/function members.

To make the module visible to SDK applications, the module developer must declare it by filling the module descriptor **DLLExportTable**. The module developer needs to tell the SDK how to create the **PXCMath** class instance. The **CreateInstance** function does just that. The **DLLExportTable** structure is a linked list structure. It is possible to implement multiple interfaces and declare them through the linked list.

Finally, the module developer needs to build the **PXCMath** class implementation and the declaration table into a shared library and register it as described in the Module Discovery Process section.

The application can access the **PXCMath** implementation as illustrated in Example 3.

```
class Math:public PXCBaseImpl<PXCMath> {
public:
       virtual pxcStatus PXCAPI Add(int i1, int i2, int *o) {
              *o=i1+i2;
              return PXC_STATUS_NO_ERROR;
       }
       static pxcStatus CreateInstance(PXCSession *session, PXCScheduler *sch,
PXCAccelerator *accel, PXCSessionService::DLLExportTable *table, pxcUID cuid,
PXCBase **instance) {
              *instance=new Math;
              retun PXC_STATUS_NO_ERROR;
       }
};

PXCSessionService::DLLExportTable __declspec(dllexport) mathImplDesc={
    0,
    Math::CreateInstance,
    PXCSessionService::SUID_DLL_EXPORT_TABLE,
    {
         PXCSessionService::IMPL_GROUP_USER,        /* group              */
         PXCMath::IMPL_SUBGROUP_MATH,               /* subgroup           */
         0,                                         /* algorithm          */
         PXC_UID('M','A','T','H'),                  /* iuid               */
         { 1, 0 },                                  /* version            */
         PXCAccelerator::ACCEL_TYPE_EMULATION,      /* acceleration       */
         100,                                       /* merit              */
         0x1000,                                    /* vendor             */
         { PXCMath::CUID, 0, 0 },                   /* exposed interfaces */
         {0}                                        /* reserved           */
    }
};
```

Example 2: The PXCMath Implementation

```
session→CreateImpl(PXCMath::CUID,&math);
```

Example 3: Accessing the PXCMath implementation in an Application

Module Discovery Process

When the application calls the **`PXCSession_Create`** function to create an SDK session, the function employs a simple discovery process to find out what SDK modules are available on the system. The application can then use the information to locate a module and create an instance of the module. See Appendix C for the discovery process for Microsoft* Windows*. The modules loaded by this discovery process are system-wide modules that any application can use.

Alternatively, if a module is not one of the pre-installed system-wide modules, the application can still load it in one of two ways:

- If the module is built as a standalone shared library file, the application can load the module using the **`LoadImplFromFile`** function after the application creates an SDK session.

- If the module is part of the application (statically linked into the application,) the application can use the **LoadImpl** function from the **PXCSessionService** interface to load the module at runtime, as illustrated in Example 4.

```
void LoadMyStaticModule(PXCSession *s, DLLExportTable *t) {
       s→DynamicCast<PXCSessionService>()→LoadImpl(t);
}
void UnloadMyStaticModule(PXCSession *s, DLLExportTable *t) {
       s→DynamicCast<PXCSessionService>()→UnloadImpl(t);
}
```

Example 4: Managing Statically Linked Module

Asynchronous Execution

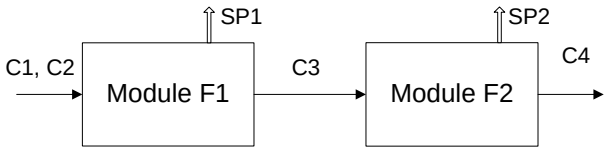
The **PXCMath** interface as in Example 1 is a simplification of any "real" SDK interfaces. The function **Add** blocks execution while doing its work. This is not desired as the application works on multiple modality operations. It is recommended to design the SDK interface to be asynchronous. Example 5 is the redefined **PXCMathAsync** interface. The **AddAsync** function takes inputs and outputs as pointers and returns a Synchronization Point (SP) so that the application can synchronize the execution result at a later time (together with other asynchronous executions).

```
class PXCMathAsync:public PXCBase {
public:
       PXC_CUID_OVERWRITE(0x87654321);
       virtual pxcStatus PXCAPI AddAsync(int *i1, int *i2, int *o,
                                         PXCScheduler::SyncPoint **sp)=0;
};
```

Example 5: The PXCMathAsync Interface

The **PXCSchedulerService** class (exposed by the core framework) provides additional functions for asynchronous pipeline implementation. This section describes the basic concepts of implementing asynchronous execution in a module.

Consider an asynchronous pipeline that contains module **F1** and **F2**, as illustrated in Example 6. Module **F1** takes **C1** and **C2** as input and produces **C3 as output**. Module **F2** takes **C3** as input and produces **C4 as output**. Since these modules run asynchronously, they return SP **SP1** and **SP2**, respectively.



Example 6: Asynchronous Execution Pipeline

When the application calls module **F1**, module **F1** should implement the following procedure, as illustrated in Example 7:

- Mark the output **C3** as execution in progress, by using the **MarkOutputs** function. This blocks any subsequent asynchronous execution that uses **C3** as input.
- Request the SDK to call back when **C1** and **C2** are ready, by using the **RequestInputs** function. At this time, since **C1** and **C2** are not ready, module F1 can only save the **C1** and **C2** pointers. The callback function (**F1C**) should derive from the scheduler **Callback** class.
- Return an SP created from the **CreateSyncPoint** function to the application.

```
scheduler_service→MarkOutputs(1,&C3,PXC_STATUS_EXEC_INPROGRESS);
PXCSchedulerService::Callback *f1c=new F1C(…);
void *inputs[]={C1, C2};
scheduler_service→RequestInputs(2,inputs,f1c);
PXCSchedule::SyncPoint *sp;
scheduler_service→CreateSyncPoint(1,&C3,&sp);
```

Example 7: Asynchronous Function Implementation Pseudo Code

Similarly, module **F2** implements the following procedure:

- Mark the output **C4** as execution in progress, by using the **MarkOutputs** function. This blocks any subsequent asynchronous execution that uses **C4** as input.
- Request the SDK to call back when **C3** is ready, by using the **RequestInputs** function. At this time, since **C3** is not ready, module F2 can only save the **C3** pointer. The callback function (**F2C**) should derive from the scheduler **Callback** class.
- Return the SP created from the **CreateSyncPoint** function to the application. Add a reference to the SP (by calling the **AddRef** function) so that the SP is persistent until the callback completes.

With the above procedures, the SDK will callback **F1C** when **C1** and **C2** are ready, and **F2C** when **C3** is ready.

The callback function **F1C** must incorporate the following steps, as illustrated in Example 8:

- If the callback input **status** is an error (**status < PXC_STATUS_NO_ERROR**), the upstream module that generates **C1** and **C2** is aborted. The function **F1C** should abort as well by completing the abort procedure:

    - Mark the output **C3** as aborted by using the **MarkOutputs** function with the status code **PXC_STATUS_EXEC_ABORTED**.
    - Mark the SP status as aborted by using the **SignalSyncPoint** function with the status code **PXC_STATUS_EXEC_ABORTED**.
- The function **F1C** produces **C3** based on **C1** and **C2**.
- Mark the output **C3** as complete by using the **MarkOutputs** function with the appropriate status code.
- Mark the SP as complete by using the **SignalSyncPoint** function with the appropriate status code.
- Finally, this function **F1C** should delete its SP and instance.

```
void F1C::Run(pxcStatus status) {
    if (status<PXC_STATUS_NO_ERROR) {
       // abort because upstream did not finish.
       myStatus=PXC_STATUS_EXEC_ABORTED;
    } else {
       // produce C3 from C1 and C2.
       myStatus=PXC_STATUS_NO_ERROR;
    }
    scheduler_service→MarkOutputs(1,&C4,myStatus);
    sp->SignalSyncPoint(myStatus);
    delete sp;
    delete this;
}
```

Example 8: Asynchronous Function Callback Pseudo Code

Similarly, the callback function **F2C** must incorporate the following steps:

- If the callback input **status** is an error (**status<PXC_STATUS_NO_ERROR**), the upstream module that generates **C3** is aborted. The function **F2C** should abort as well by completing the abort procedure:
    - Mark the output **C4** as aborted by using the **MarkOutputs** function with the status code **PXC_STATUS_EXEC_ABORTED**.
    - Mark the SP status as aborted by using the **SignalSyncPoint** function with the status code **PXC_STATUS_EXEC_ABORTED**.
- The function **F1C** produces **C4** based on **C3**.
- Mark the output **C4** as complete by using the **MarkOutputs** function with the appropriate status code.
- Mark the SP as complete by using the **SignalSyncPoint** function with the appropriate status code.
- Finally, this function **F2C** should delete its SP and instance.

The SDK provides the **PXCSmartAsyncImpl** set of template interfaces to simplify asynchronous execution implementations. The template interfaces convert an asynchronous execution to an equivalent synchronous execution.

Example 9 shows the implementation of an asynchronous **Invert** function. The function saves all input and output pointers and requests a callback when the input parameters are ready, at the same time, blocking any potential executions on the function outputs. Then in the callback

function implementation, it computes the outputs and signals the application for completion as well as any subsequent executions waiting for the produced results.

Using the **PXCSmartAsyncImpl** set of templates, the asynchronous execution code can be simplified as illustrated in Example 10. The **InvertAsync** function simply invokes the **SubmitTask** function to handle all asynchronous handling code, and the **InvertSync** function is a synchronous function that implements the actual logic of producing the output from the input.

```
class Math:public PXCBaseImpl<PXCMathAsync> {
public:
    virtual pxcStatus PXCAPI Invert(pxcF32 *v, pxcF32 *o, PXCScheduler::SyncPoint **sp);
protected:
    class INVERT:public PXCBaseImpl<PXCSchedulerService::Callback> {
    public:
        INVERT(Math *math, PXCScheduler::SyncPoint *sp, pxcF32 *v, pxcF32 *o);
        virtual void PXCAPI Run(pxcStatus status);
    protected:
        PXCSchedulerService::SyncPointService *sp;
        pxcF32 *v, *o;
        Math *math;
    };
    PXCSchedulerService *scheduler;
};
Math::INVERT::INVERT(Math *math, PXCScheduler::SyncPoint *sp, pxcF32 *v, pxcF32 *o) {
    this->sp=(PXCSchedulerService::SyncPointService*)sp->
                    DynamicCast(PXCSchedulerService::SyncPointService::CUID);
    this->sp->AddRef(); this->math=math; this->v=v; this->o=o;
}

void Math::INVERT::Run(pxcStatus status) {
    if (status>=PXC_STATUS_NO_ERROR) {
        *o= (pxcF32)1.0/(*v);
        status=PXC_STATUS_NO_ERROR;
    } else {
        status=PXC_STATUS_EXEC_ABORTED;
    }
    math->scheduler->MarkOutputs(1,(void**)&o,status);
    sp->SignalSyncPoint(status);
    delete sp; delete this;
}

pxcStatus Math::InvertAsync(pxcF32 *v, pxcF32 *o, PXCScheduler::SyncPoint **sp){
    scheduler->CreateSyncPoint(1,(void**)&o,sp);
    scheduler->MarkOutputs(1,(void**)&o,PXC_STATUS_EXEC_INPROGRESS);
    scheduler->RequestInputs(1,(void**)&v,new INVERT(this,*sp,v,o));
    return PXC_STATUS_NO_ERROR;
}
```

Example 9: Implementation of An Asynchronous Invert Function

```
class Math:public PXCBaseImpl<PXCMath> {
public:
    virtual pxcStatus PXCAPI InvertAsync(pxcF32 *v, pxcF32 *o, PXCScheduler::SyncPoint
**sp);
    virtual pxcStatus PXCAPI InvertSync(pxcF32 *v, pxcF32 *o);
protected:
    PXCScheduler *scheduler;
};

pxcStatus Math::InvertSync(pxcF32 *v, pxcF32 *o) {
    *o= (pxcF32)1.0/(*v);
    return PXC_STATUS_NO_ERROR;
}

pxcStatus Math::InvertAsync(pxcF32 *v, pxcF32 *o, PXCScheduler::SyncPoint **sp) {
    return PXCSmartAsyncImpl<MathAsync,pxcF32,pxcF32>::
                    SubmitTask(v,o,sp,this,scheduler,&MathAsync::InvertSync);
}
```

Example 10: PXCSmartAsyncImpl Implementation of An Asynchronous Invert Function

There are different variations of the **PXCSmartAsyncImpl** templates. The template **PXCSmartAsyncImpl** is used for asynchronous functions that produce a single output from a single input. The templates **PXCSmartAsyncImplIsOt** are for asynchronous functions that generate **s** number of outputs from **t** number of inputs.

Module developers should use caution when using the multiple inputs to multiple outputs templates. The **PXCSmartAsyncImpl** templates do not invoke the synchronous callback function before all inputs are ready and do not signal the application or subsequent executions until all outputs are generated by the synchronous callback function. Thus these templates do not work well (in terms of execution efficiency) in a complex pipeline where the module can generate some outputs from a subset of inputs or the module must signal each output individually to improve the pipeline performance. To workaround, the module developer may need to concatenate two or more of these templates together to achieve the best efficiency.

Hardware Acceleration

Hardware acceleration may be necessary for some SDK modules to achieve certain performance or power requirements. The SDK provides the following means to facilitate hardware acceleration development while leaving the actual implementation to the SDK module developers:

- The **acceleration** field of the **PXCSession::ImplDesc** structure declares the acceleration framework that a module supports. The application can choose those modules that support hardware acceleration for better performance.
- When there are multiple hardware accelerated modules in a pipeline, the application may want to ensure that they work under the same acceleration context. The **PXCAccelerator** interface, passed as part of the module **CreateInstance** parameters,

manages the acceleration context. The application can use the **SetDevice** and **SetHandle** functions to set the acceleration context and the SDK module uses the **QueryDevice** and **QueryHandle** function to retrieve the acceleration context.

- The **PXCImage** and **PXCAudio** interfaces abstract image and audio storage. A hardware accelerated module can export its native data storage to the **PXCImage** and **PXCAudio** instances. The subsequent modules (in the pipeline) that use the same acceleration context can import the data without any data copying performance penalty.

Interface and Function Reference

This section describes SDK functions and their operations.

In each function description, only commonly used status codes are documented. The function may return additional status codes in certain cases. See the **pxcStatus** enumerator for a list of all status codes.

PXCSchedulerService::Callback

The `Callback` interface defines an asynchronous execution callback function. The module developer needs to implement this interface.

The `callback` interface provides the following member functions:

Member Functions	Description
Run	The asynchronous execution callback function.

Run

Syntax

```
pxcStatus Run(pxcStatus status);
```

Parameters

`status`	The upstream processing status.

Description

The SDK calls this function for asynchronous execution when the requested inputs are ready. The module developer should abort the execution if the upstream execution is unsuccessful (**status<PXC_STATUS_NO_ERROR**).

The module developer should delete the instance after callback.

Return Status

The SDK ignores the return status of this function.

Change History

This function was introduced in SDK API 1.0.

PXCSchedulerService::SyncPointService

The `SyncPointService` interface provides additional functions for the module developer to implement asynchronous execution. The module developer can obtain a pointer to this interface by using the **`DynamicCast`** function from the **`SyncPoint`** instance.

The `SyncPointService` interface provides the following member functions:

Member Functions	Description
`AddRef`	Add a reference count to the SP.
`SignalSyncPoint`	Set the SP execution status.

AddRef

Syntax

void `AddRef(void);`

Description

This `AddRef` function increases the SP reference count.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

SignalSyncPoint

Syntax

`pxcStatus` `SignalSyncPoint(`**`pxcStatus`** `status);`

Parameters

`status`	The SP execution status.

Description

This `SignalSyncPoint` function sets the SP execution status.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

PXCSchedulerService

The `PXCSchedulerService` interface provides additional functions for the module developer to implement asynchronous execution. The application can obtain a pointer to this interface by using the **`DynamicCast`** function from the **`PXCScheduler`** instance.

The `PXCSchedulerService` interface provides the following member functions:

Member Functions	Description
`CreateSyncPoint`	Create an SP instance.
`RequestInputs`	Request callback when the desired inputs are ready.
`MarkOutputs`	Mark outputs as ready or in progress.

CreateSyncPoint

Syntax

```
pxcStatus CreateSyncPoint(pxcU32 noutput, void** outputs, SyncPoint **sp);
```

Parameters

`noutput`	The number of outputs.
`outputs`	The array of output pointers
`sp`	The SP instance, to be returned.

Description

This `CreateSyncPoint` function creates an instance of the SP. The parameters **`noutput`** and **`outputs`** describe the outputs of an asynchronous function. The SDK associates the SP with the outputs so that the application can query the execution status of each output.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

MarkOutputs

Syntax

```
pxcStatus MarkOutputs(pxcU32 noutput, void** outputs, pxcStatus sts);
```

Parameters

`noutput`	The number of outputs.
`outputs`	The array of output pointers
`sts`	The output status.

Description

This `MarkOutputs` function sets the output execution status.

If the outputs are not ready, the module developer should set the output status as **`PXC_STATUS_EXEC_INPROGRESS`**.

If the outputs are ready, the module developer should set the output status as **`PXC_STATUS_NO_ERROR`**. Note that the module developer needs to additionally set the SP status to **`PXC_STATUS_NO_ERROR`** after completing all outputs.

If the execution is aborted due to upstream errors, the module developer should set the output status to **`PXC_STATUS_EXEC_ABORTED`**. Note that the module developer needs to additionally set the SP status to **`PXC_STATUS_EXEC_ABORTED`**.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

RequestInputs

Syntax

```
pxcStatus RequestInputs(pxcU32 ninput, void** inputs, Callback *cb);
```

Parameters

`ninput`	The number of inputs.
`inputs`	The array of input pointers.
`cb`	The scheduler callback instance.

Description

This `RequestInputs` function queues the callback request to the SDK scheduler. The SDK scheduler will call back when the specified inputs are available (by upstream SDK modules in an asynchronous execution pipeline.)

The module developer must increase the SP reference after this function and delete the SP after the callback is complete to persist the SP during asynchronous execution.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.[s

Change History

This function was introduced in SDK API 1.0.

PXCSerializableService

The `PXCSerializableService` interface provides functions to serialize an interface implementation and recreates such implementation.

The `PXCSerializableService` interface provides the following member functions:

Member Functions	Description
QueryProfile	Retrieve the configuration parameters.

SetProfile	Set the configuration parameters.

QueryProfile

Syntax

```
pxcStatus QueryProfile(ProfileInfo *pinfo, pxcBYTE *data);
```

Parameters

pinfo	The configuration parameters to be returned.
data	Optional pointer to retrieve initialization data.

Description

This function returns the serialization implementation configuration parameters. To retrieve the initialization data, the application calls this function twice: the first time with data pointer **NULL** to retrieve the buffer size, and the second time with the allocated buffer.

Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

SetProfile

Syntax

```
pxcStatus SetProfile(ProfileInfo *pinfo, pxcBYTE *data);
```

Parameters

pinfo	The configuration parameters to be returned.
data	The initialization buffer pointer.

Description

This function initializes the serializable implementation.

Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

PXCSessionService

The PXCSessionService interface provides some additional functions for module developers to manage modules. The application can obtain a pointer to the PXCSessionService interface by calling the **DynamicCast** function from the SDK session instance.

The PXCSessionService interface provides the following member functions:

Member Functions	Description
QueryImplEx	Query a module including core services.
LoadImpl	Add a run-time module to the session.
UnloadImpl	Remove a run-time module from the session.
LockSession	Lock the session for exclusive access.
UnlockSession	Unlock the session from exclusive access.

LoadImpl

Syntax

```
pxcStatus LoadImpl(DLLExportTable *table);
```

Parameters

Table	The implementation description table.

Description

This LoadImpl function loads a module implementation from memory at run time.

Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

LockSession

Syntax

```
pxcStatus LockSession(void);
```

Description

This `LockSession` function locks the session for exclusive access. This function blocks if there is already a lock to the session.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

QueryImplEx

Syntax

```
pxcStatus QueryImplEx(ImplDesc *template, pxcU32 idx, DLLExportTable
**table, void ***storage);
```

Parameters

`template`	The search template: for any field in the structure, a zero means matching any values. For bit-OR'ed fields, a non-zero value means matching the corresponding bits.
`idx`	The zero-based index.
`table`	The implementation description table, to be returned.
`storage`	The module local storage, to be returned.

Description

This `QueryImplEx` function is an extension to the **`QueryImpl`** function of the **`PXCSession`**

class. Besides searching public module implementations, this function also searches and returns core service implementations, such as **PXCAccelerator** and **PXCScheduler**. The function also returns each module's local storage. The SDK session allocates the local storage of size **sizeof(void*)** for each module regardless how many instances a module may have.

Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.
PXC_STATUS_ITEM_UNAVAILABLE	There is no implementation match.

Change History

This function was introduced in SDK API 1.0.

UnloadImpl

Syntax

```
pxcStatus UnloadImpl(DLLExportTable *table);
```

Parameters

table	The implementation description table.

Description

This `UnloadImpl` function removes a module implementation at run time.

Return Status

PXC_STATUS_NO_ERROR	The function returned successfully.
PXC_STATUS_ITEM_UNAVAILABLE	Failed to locate the specified module.

Change History

This function was introduced in SDK API 1.0.

UnlockSession

Syntax

```
pxcStatus UnlockSession(void);
```

Description

This `UnlockSession` function releases the exclusive lock on the session.

Return Status

`PXC_STATUS_NO_ERROR`	The function returned successfully.

Change History

This function was introduced in SDK API 1.0.

PXCSmartAsyncImpl

The **`PXCSmartAsyncImpl`** set of template interfaces convert an asynchronous execution to an equivalent synchronous execution to simplify asynchronous execution implementation.

PXCSmartAsyncImpl<T,T1,T2>::SubmitTask

Syntax

```
      pxcStatus SubmitTask(T1* input, T2* output, PXCScheduler::SyncPoint **sp,
T* instance, PXCScheduler *scheduler, pxcStatus (PXCAPI T:*)(T1*,T2*) syncFunc,
pxcStatus (PXCAPI T:*)(pxcStatus) abortFunc=0, const pxcCHAR *name=0);
```

Parameters

`input`	The input parameter.
`output`	The output parameter.
`sp`	The SP, to be returned.
`instance`	The instance of the callback function **`syncFunc`**.
`scheduler`	The **`PXCScheduler`** instance.
`syncFunc`	The callback function pointer for performing the module operations when the input data is ready.
`abortFunc`	The optional callback function pointer for error handling upon upstream abort errors.
`name`	The optional task name string.

Description

This **SubmitTask** function handles the asynchronous execution procedure for the single input and single output case. When the input is ready, the scheduler will callback the **syncFunc** function with the saved input and output parameters. When the processing is aborted due to upstream module errors, the scheduler will callback the **abortFunc** function if available.

The callback function **syncFunc** takes the form: **pxcStatus PXCAPI T::Name(T1 *input, T2 *output)**. The return value is passed to the synchronization status.

The callback function **abortFunc** takes the form: **pxcStatus PXCAPI T::Name(pxcStatus)**. The input is the upstream module status. The return value is passed to the synchronization status.

Return Status

Any return status should be reported back to the application.

Remarks

There are a few variations of this template. The `SubmitTask` function in the **PXCSmartAsyncImplIsOt** template takes **s** number of inputs and **t** number of outputs.

PXCSmartAsyncImplIxOy<T,Ti,x,To,y>::SubmitTask

Syntax

```
pxcStatus SubmitTask(Ti* inputs[], pxcU32 ninputs, To* outputs[], pxcU32
noutputs, PXCScheduler::SyncPoint **sp, T* instance, PXCScheduler *scheduler,
pxcStatus (PXCAPI T:*)(Ti**,To**) syncFunc,pxcStatus (PXCAPI T:*)(pxcStatus)
abortFunc=0, const pxcCHAR *name=0);
```

Parameters

`inputs`	The array of inputs.
`ninputs`	The number of actual inputs.
`output`	The array of outputs.
`noutputs`	The number of actual outputs.
`sp`	The SP, to be returned.
`instance`	The instance of the callback function **syncFunc**.
`scheduler`	The **PXCScheduler** instance.

`syncFunc`	The callback function pointer.
`abortFunc`	The optional callback function pointer for error handling upon upstream abort errors.
`name`	The optional task name string.
`x`	The maximum number of inputs
`y`	The maximum number of outputs

Description

This **`SubmitTask`** function handles the asynchronous execution procedure for the single input and single output case. When the input is ready, the scheduler will callback the **`syncFunc`** function with the saved input and output parameters. When the processing is aborted due to upstream module errors, the scheduler will call back the **`abortFunc`** function if available.

The callback function **`syncFunc`** takes the form: **`pxcStatus PXCAPI T::Name(Ti *inputs[], To *outputs[])`**. The return value is passed to the synchronization status.

The callback function **`abortFunc`** takes the form: **`pxcStatus PXCAPI T::Name(pxcStatus)`**. The input is the upstream module status. The return value is passed to the synchronization status.

Return Status

Any return status should be reported back to the application.

Structure Reference

In the following structure references, all reserved fields must be zero.

PXCSessionService::DLLExportTable

Definition

```
struct DLLExportTable {
      DLLExportTable    *next;
      pxcStatus (PXCAPI *createInstance)(PXCSession *session, PXCScheduler
*scheduler, PXCAccelerator *accel, DLLExportTable *table, pxcUID cuid,
PXCBase **instance);
      pxcUID            suid;
      ImplDesc          desc;
};
```

Description

The **DLLExportTable** structure describes a module implementation and its constructor function. The developer can link multiple module implementations together with the **next** field.

Members

next	Point to the next implementation description table.
createInstance	The function pointer to create an instance of the implementation.
suid	Must be **PXCSessionService::SUID_DLL_EXPORT_TABLE**.
desc	The module descriptor.

Change History

This structure is introduced in SDK API 1.0.

PXCSerializableService::ProfileInfo

Definition

```
struct ProfileInfo {
      PXCSession::ImplDesc    implDesc;
      pxcU32                  dataSize;
      pxcU32                  reserved[7];
};
```

Description

The **ProfileInfo** structure describes parameters of a serializable interface implementation.

Members

implDesc	The serializable implementation description table.
dataSize	The data size in bytes of the initialization data of a serializable implementation.

Change History

This structure is introduced in SDK API 1.0.

Appendixes

Appendix A: SDK API Conventions

A typical SDK interface is illustrated in Example 11:

```
class PXCMyInterface:public PXCBase {
public:
       PXC_CUID_OVERWRITE(0x12345678); // my unique interface identifier

       // my configurations
       struct ProfileInfo {
              ….
       };

       // configuration inquiry and set working configuration
       virtual pxcStatus PXCAPI QueryProfile(pxcU32 idx, ProfileInfo *pinfo)=0;
       virtual pxcStatus PXCAPI SetProfile(ProfileInfo *pinfo)=0;

       // Now do something asynchronously
       virtual pxcStatus PXCAPI ProcessAsync(…, PXCScheduler::SyncPoint *sp)=0;
};
```

Example 11: Typical SDK Interface Design

Naming Conventions

- All classes, functions, and structures in the global name space use "PXC" or "pxc" as a prefix.
- Minimize creating classes, functions, and structures in the global name space. Put classes, functions, and structures in their respective class scopes.
- The class and function names start with a capital letter.
- The variable names start with a lower-case letter.
- If a class, function, or variable name contains multiple words, usually the SDK capitalizes the first letter of the second word and thereafter.

```
class MyClass: public PXCBase {
public:
      struct MyStruct {
            MyStruct    *myField1;
            pxcU32      myField2;
      };
};
```

- The macros and enumerator definitions use all capital letters with words separated by an underline.

```
#define PXC_DEFINE_UID(…) …
```

Interface and Member Function Conventions

- Each SDK interface is a C++ pure virtual class derived from the **PXCBase** interface. Each SDK callback interface is a C++ pure virtual class derived from the **PXCBaseCallback** interface. The SDK interface must have a unique class identifier. Use the **PXC_CUID_OVERWRITE** macro to define the class identifier.
- Each SDK interface must contain pure virtual functions only, with or without default implementation. No constructor or destructor.
- Do not use exception handling or dynamic cast in interface definitions or module implementations as these features do not reliably work across multiple compilers.
- Most SDK functions should return a status code, **pxcStatus**, unless the functions are simple enough that they return some internal variables without any processing.

```
virtual pxcStatus   PXCAPI  SetTimeStamp(pxcU64 ts)=0;
virtual pxcU64      PXCAPI  QueryTimeStamp(void)=0;
```

- For bit-OR'ed field enumerators, the SDK usually defines them as a type definition of **pxcEnum** and then defines the values in a separate anonymous enumerator. The SDK defines an enumerator directly if the SDK uses the values directly without calculation.

```
typedef pxcEnum AccelType;
enum {
      ACCEL_TYPE_ANY              = 0,
      ACCEL_TYPE_EMULATION        = 0x00000001,
      ACCEL_TYPE_VIA_DX9          = 0x00000002,
};

enum HandleType {
      HANDLE_TYPE_DX9_DEVICE     =   0x00000002,
      HANDLE_TYPE_DX11_DEVICE    =   0x00000008,
};
```

- For consistency, use the following two functions for module configuration query and setup:

```
virtual pxcStatus PXCAPI QueryProfile(pxcU32 pidx, ProfileInfo
*pinfo);
virtual pxcStatus PXCAPI SetProfile(ProfileInfo *pinfo);
```

Data and Control Flow Conventions

- Use the **PXCImage** and **PXCAudio** interfaces to abstract image and audio buffers. These buffers may be in system memory buffers or in OS-specific surfaces.

- Any SDK function that performs anything more than trivial tasks should be asynchronous. The function should return immediately with an SP.
- Design asynchronous I/O functions so that the functions can be chained into an asynchronous pipeline for better performance. A counter example is that if there are multiple faces in a picture, it is not a good design if the asynchronous function only detects a single face. The application must make multiple calls on the picture. The application cannot determine beforehand how many loops are needed. Instead, design the function to detect all faces at once. Any subsequent asynchronous functions should also process all faces.
- Asynchronous functions and asynchronous pipelines work best if, with any input data, each function generates some output that can be processed by subsequent functions. It is not a good design if an asynchronous function requires more inputs (or worse, irregular amount of inputs) to generate an output. Consider a callback design for this type of pattern of operations.

Documentation Conventions

Follow the documentation styles of this document.

Appendix B: Singleton Module

The SDK exposes all modules as class interfaces and provides the session function **`CreateImpl`** to create instances of the classes. This implies that all modules work as multiple instances. This section describes a way to create a singleton module within the SDK session. The application will retrieve the same module instance from multiple **`CreateImpl`** function calls.

```
static void** QueryStorage(PXCSessionService *ss) {
	PXCSession::ImplDesc desc;
	memset(&desc,0,sizeof(desc));
	desc.iuid=Math::IUID;
	void** storage=0;
	ss->QueryImplEx(&desc,0,0,&storage);
	return storage;
}

class Math:public PXCBaseImpl<PXCMath> {
public:
	Math(PXCSessionService *ss) { ref=1; this->ss=ss; }
	virtual pxcStatus PXCAPI Add(int i1, int i2, int *o) {
		*o=i1+i2;
		return PXC_STATUS_NO_ERROR;
	}
	virtual pxcStatus PXCAPI Release(void) {
		ss→LockSession();
		int ref= --ref;
		ss→UnlockSession();
		if (ref) return PXC_STATUS_NO_ERROR;
		return PXCBaseImpl<PXCMath>::Release();
	}
	void IncreaseReference(void) { ref++; }
protected:
	PXCSessionService *ss;
	int ref;
};
static pxcStatus CreateInstance(PXCSession *s, PXCScheduler *sch, PXCAccelerator
*accel, PXCSessionService::DLLExportTable *table, pxcUID cuid, PXCBase
**instance) {
	PXCSessionService *ss=(PXCSessionService*)s→
				DynamicCast(PXCSessionService::CUID);
	ss→LockSession();
	Math **ppmath=(Math **)QueryStorage(ss);
	if (*ppmath) {
		ppmath→IncreaseReference();
	} else {
		*ppmath=new Math(ss);
	}
	ss→UnlockSession();
	*instance= *ppmath;
	retun PXC_STATUS_NO_ERROR;
}
```

Example 12: The PXCMath Singleton Implementation

Continuing with the **PXCMath** example, as illustrated in Example 12, the module developer needs to do two things differently as follows:

- During construction, the module developer must ensure that only a single instance is created. The **CreateInstance** in Example 12 uses the local storage provided by the SDK session to store the first instance. Any subsequent **CreateInstance** call increases the reference counter and returns the same instance.
- During destruction, the module developer decreases the reference counter and only deletes the instance when the reference counter reaches zero. To do so, the module developer needs to overload the **PXCBaseImpl::Release** function, which is an auxiliary function that invokes the module destructor.

Appendix C: Module Discovery Under Microsoft* Windows*

The SDK lists system-wide modules as a set of subkeys (whose names are ignored) under the registry path **HKLM/Software/Intel/PCSDK/Dispatch**. Under each module subkey, the SDK lists the module description parameters as in **PXCSession::ImplDesc** and Table 1. When an application creates a session instance, the SDK reads the module values from the registry and uses these values for module search. The SDK loads the module DLL only when the application creates a module instance.

Key Value	Data Type	Description
group	**DWORD**	The module group identifier.
subgroup	**DWORD**	The module subgroup identifier.
algorithm	**DWORD**	The module algorithm identifier.
iuid	**DWORD**	The module unique implementation identifier.
versionMajor	**DWORD**	The module version major number.
versionMinor	**DWORD**	The module version minor number.
acceleration	**DWORD**	The module acceleration identifier.
merit	**DWORD**	The module merit value.

vendor	**DWORD**	The module vender identifier.
cuid	**DWORD**	The first module interface identifier: **PXCSession::ImplDesc::cuids[0].**
cuid2	**DWORD**	The second module interface identifier: **PXCSession::ImplDesc::cuids[1].**
cuid3	**DWORD**	The third module interface identifier: **PXCSession::ImplDesc::cuids[2].**
cuid4	**DWORD**	The fourth module interface identifier: **PXCSession::ImplDesc::cuids[3].**
friendlyName	**STRING**	The module friendly name. Limited to 255 characters.
path	**STRING**	The module DLL with the absolute path.

Table 1: Registry Settings For a System-Wide Module