Intel® Atom™ Developer Program SDK Developer Guide

January 2010

Document version 0.9 -rev (2)

Notice:

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppels 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. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice

The API and software 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.

This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

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 ordering 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 website at http://www.intel.com.

Copyright © 2009 Intel Corporation.

* Third party names and brands may be claimed as the property of others.

Contents

Introduction 5
Document Overview 5
Conventions 5
Overview 6
Intel® Atom™ Developer Program 6
Software Development Kit (SDK) 7
Application Services Libraries 7
Application Test and Debug Service 7
Documentation 8
Development Process Integration (FUTURE) 8
Developer Portal Integration (FUTURE) 8
Developer Store Integration (FUTURE) 8
Package Utility (FUTURE) 8
Submit Utility (FUTURE) 8
Runtime Interaction 8
Hello World 8
C Language API 8
C++ Framework 9
Getting Started 10
Get an Application or Component ID 10
C Language API 10
C++ Framework 10
Compile and Link 10
Compiling 10
Linking 11
Debug and Test 11
Debugging 11
Testing 11
Submit the Application to the Store 12
Writing an Application 12
License Enforcement 12
C++ Framework 12
Delegate 12
Sub-classing Application 13
Application::Application 13
C Language API 13
Writing a Component 15

License Enforcement 15
C++ Framework 15
Component::Component 15
C Language API 16
Putting It All Together 16
C++ Framework 16
C Language API 18
API Versions and Levels 19
C Language API 19
C++ Framework 19
Crash Reporting 19
C Language API 20
Customizing the Crash Report 21
C++ Language Framework 22
AbstractCrashReport 24
DefaultCrashReport 24
Application Instrumentation 24
C Language API 24
C++ Framework 25
Error handling 25
C Language API 26
C++ Framework 26
Appendices 27
Crash Reporting 27
Fields 27
Source 27
Example: Crash Report 27
Test Case Matrix 28

Introduction

This document describes the Intel® Atom™ Developer Program Software Developer Kit (SDK), a software package the contains programming libraries, documentation, tools, IDE plug-ins, references, links, and so on, which allow a developer to create, test, and deploy Intel® Atom™ Developer Program applications to the Consumer Store and components to the Developer Store.

> **NOTE:** This is preliminary documentation and is subject to change.
>
> Although we do not expect the interfaces to change, a recompile/re-link maybe required prior to final release. Subsequent versions will support concurrent interface versions.
>
> This release only addresses application and component authorization and some core framework functionality. Subsequent releases will have greatly expanded functionality.
>
> This document addresses use of the C Language library and C++ Language framework and applies to both Microsoft Windows* OS and Moblin* OS.

For more details on the Intel® Atom™ Developer Program, including how to join the program, visit "http://appdeveloper.intel.com ". Although membership is not required to download and use the SDK, it is required to redistribute your applications or components.

Document Overview

This document describes important concepts and core capabilities of the SDK, and provides information on how design, develop and debug application and components.

This document contains the following chapters:

- **Overview** – Provides a high overview of the Intel® Atom™ Developer Program, the Software Development Kit, and describes the runtime interactions of the systems.
- **Hello World** – Provide an example of an application in C and in C++.
- **Getting Started** – Describes how to create, debug/test and submit an application or component
- **Writing Applications** – Details how to create applications using the C Language API and C++ Framework
- **Writing Components** - details how to create components using the C Language API and C++ Framework.
- **Putting it Together** –Shows how to use applications and components together
- **Detail Topics** – Contains that provide greater details on key features of the SDK such crash reporting, instrumentation, and so on.

Conventions

Several conventions are used in this document to indicate meaning or intent of the text.

Formatting	Meaning
Bold Text	Denotes a term or character to be type literally, such as predefined data type or function name. You must type these terms exactly as shown.
Italic TextHyp	Denotes a placeholder or variable: You must provide the actual value. For example, the statement uCloseConnection(*hConnection*) requires you to substitute values for the *hConnection* parameters.
[ ]	Encloses optional parameters

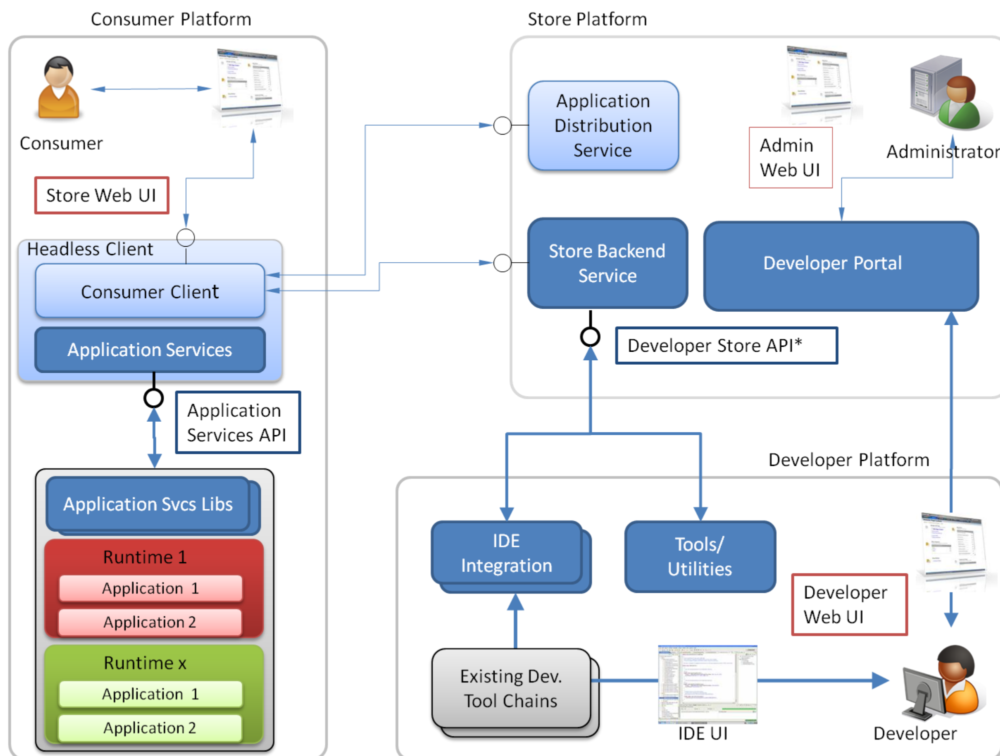
|	Separates an either/or choice.
…	Represents an omitted portion of a sample.
SMALL CAPITALS	Indicate the names of keys, key sequences, and key combinations – for example, ALT+SPACEBAR.
FULL CAPITALS	Indicate filename and paths, most type structure names (which are also bold), and constants.
`fixed width font text`	Sets off code examples and shows syntax spacing.
Hyperlink	A hyperlink to internal and external information. Use CTRL-mouse click to navigate.
Notes, warnings, callouts, and so on	Boxed text

Overview

This section provides context and background for understanding the role and use of the Sottware Development Kit (SDK).

Intel® Atom™ Developer Program

The Intel® Atom™ Developer Program SDK is used to develop applications and components for redistribution via the Intel® Atom™ Developer Program (see below).



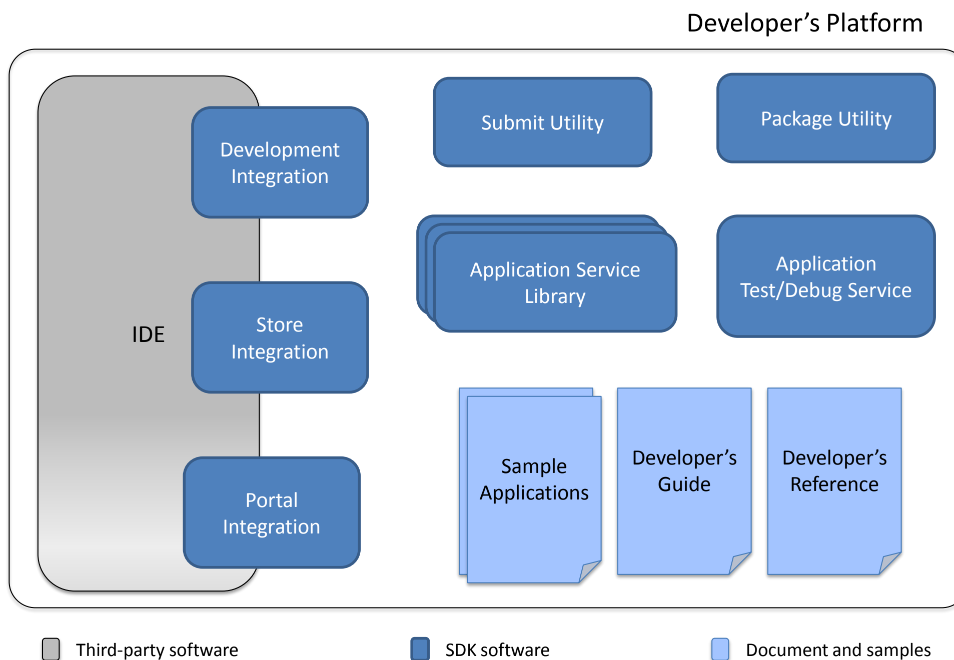
The systems and interfaces relevant to the SDK are filled dark blue in the case of systems and thick lines in the case of communication.

Software Development Kit

The goal of the SDK is twofold: first, provide programming frameworks and libraries that make the development of Intel® Atom™ Developer Program applications or components easy and fast; and provide the tools and utilities to facilitate the development process and make the interaction with the Intel® Atom™ Developer Program processes and infrastructure seamless and frictionless.

> **NOTE:** This section describes components and functionality not contained in the current release. They are labeled as **(FUTURE)** in the text

The following is a diagram of the components.



Application Services Libraries

The Application Services libraries provide you with access to Consumer Client services, such as licensing and metering; store functionality, such as in-application ratings; application/services integrations, and access to platform features.

> **NOTE:** The current version of the SDK only provides application and component Licensing and some basic utility functions.
>
> For the definitive description of the APIs and functionality contained in this release please see C Language API Reference and C++ Framework API reference.

There will be a separate implementation of the library for each supported runtime and, in the case of a native application, one for each supported operating system.

The Application Services libraries are the only parts of the SDK that are redistributed along with an application.

Application Test and Debug Service

The Application Test and Debug Service is a utility that can emulate the Application Service of the Consumer Client (see the figure above for details) as well as other services provided by

the SDK. It allows you to debug and test your application or component without requiring the full client or access to hardware, and so on

Documentation

There are three types of documentation provided with the SDK: the Developer's Guide which explains how to develop applications and components; the developer references which describe the frameworks and APIs of the Application Services library; and sample source code which shows the use of the Application Services API.

> **NOTE:** The following are planned features of the SDK. They are included for preview and information only.

Development Process Integration (FUTURE)

The Development Process integration makes it easy for you to quickly create applications or components directly within the IDE. It guides you through the steps of setting up the application, and the environment and then generates the boilerplate code for the application, including any Intel® Atom™ Developer Program SDK specific code.

Developer Portal Integration (FUTURE)

The Developer Portal Integration allows you to use portions of Developer Portal. For example you can access Forums, Dashboard, and so on, within the IDE.

Developer Store Integration (FUTURE)

The Developer Store Integration allows you to interact with the Developer Store. For example you can browse the catalog, purchase components, and so forth, within the IDE.

Package Utility (FUTURE)

You can use the Package utility to package an application for submission to the store.

Submit Utility (FUTURE)

You can use the Submit utility to submit an application to the store.

Runtime Interaction

For certain API features, e.g. licensing, crash reporting and instrumentation, communication with the back-end servers is indirect. The libraries communicate with a local Client Agent (labeled the Headless Client in the diagram above) which in turn communicates with the back end on their behalf. This allows core capabilities such as licensing to work offline, and provides a measure of process and control isolation.

> **NOTE:** Application instrumentation and crash reporting, in particular, are affected by this architecture. For example, although an application may call the Instrumentation API to collect usage statistics, it is the Client Agent and the user's preferences that determine whether the data is sent to the back end.

Hello World

> This document covers the use of the Intel® Atom™ Developer Program SDK C Language API (C Language API) and the Intel® Atom™ Developer Program SDK C++ Framework (C++ Framework). In the future, other languages and runtimes will be added.

Writing an application or a component is simple: basically you need to make some function calls to determine whether the machine is authorized to run your application, or in the case of a component, whether the application is authorized. Below are C and C++ programs demonstrating the most basic of applications, "Hello World."

C Language API

```
#include <stdio.h>
#include "adpcore.h"
int main( int argc, char* argv[] )
{
   ADP_RET_CODE ret_code;

   // Please use the application GUID obtained from the
   // Intel Atom Developers Portal or a ADP_DEBUG_APPLICATIONID
   const ADP_APPLICATIONID myApplicationID = {{
     0x00000000,0x11111111,0x11111111,0x11111111}};

   if ((ret_code = ADP_Initialize()) != ADP_SUCCESS ){
      printf( "ERROR: exiting" );
      exit( -1 );
   }
   if (( ret_code = ADP_IsAuthorized( myApplicationId )) ==
ADP_AUTHORIZED )
      printf( "Hello World" );
   else
      printf( "Not authorized to run" );
   exit 0;
}
```

C++ Framework

```
#include "adpcppf.h"
#include <iostream>

const ApplicationId myApplicationID(
      ADP_DEBUG_APPLICATIONID);

int main(int argc, char** argv) {
      Application * myApplication = NULL;
      try {
            myApplication = new Application(myApplicationID);
      } catch (AdpException& e) {
            cout << "Caught exception in application: " <<
                  e.what() << endl;
      }
}
```

In the C version, the **ADP_IsAuthorized()** function checks with the licensing system to see if the current machine is authorized to run the application identified by **ADP_APPLICATIONID**. In the case of C++, authorization is automatically handled in constructor of the **application** class

Next sections explain how to develop applications and components, how to handle errors and describe the lifecycle of applications and components.

Getting Started

This section covers getting the SDK installed and general process of creating, debugging and releasing an application or component in the Store.

Get an Application or Component ID

The first step in creating application or component is to get an application ID or component ID. Applications and components are uniquely indentified within the Intel® Atom™ Developer Program using a 128-bit number called an application ID or component ID respectively. The IDs are used by the licensing system to determine whether the machine is authorized to run the application or the application is authorized to use a component.

There are two types of IDs: Debug and Production. An application or component compiled with a Debug IDs can be run and debugged using the Application Test and Debug Service (ATDS), but cannot be uploaded to the Store, if they are not recognized as licensed. Applications and components with Production IDs on the other hand cannot run using the Application Test and Debug Service, but can be uploaded to the store and run on licensed machines.

> **NOTE:** You are free to create and debug applications and components using the DEBUG application and component Id, but you must register with the Intel® Atom™ Developer Program ("http://appdeveloper.intel.com") to generate application and component Ids to redistribute applications

To get started, just use the predefined Debug IDs provided by the SDK. For example:

C Language API

```
#include <stdio.h>
#include "adpcore.h"
int main( int argc, char* argv[] )
{
   ADP_RET_CODE ret_code;
   const ADP_APPLICATIONID myApplicationID = ADP_DEBUG_APPLICATIONID;
. . .
```

C++ Framework

```
#include "adpcppf.h"
#include <iostream>

const ApplicationId myApplicationID(ADP_DEBUG_APPLICATIONID);

int main( int argc, char** argv )
{
      Application * myApplication = NULL;
. . .
}
```

Compile and Link

To build the application or component, you need to include a header and add a library to the link command line.

Compiling

Source code for applications and components must include the following header files.

Language/Runtime	Header file
C language	adpcore.h
C++ Language	adpcppf.h

To avoid name collision the C Language API prepends ADP_ to all identifiers. In the case of the C++ Framework, a namespace *com::intel::adp* is used.

Linking

Applications and components must be statically linked with the Application Service library that corresponds to their target runtime. The table below lists the static libraries.

Language/Runtime	Debug	Release
C language	adpcored.lib, psapi.lib, shlwapi.lib	adpcore.lib, psapi.lib, shlwapi.lib
C++ Language	adpcored.lib, adpcppfd.lib, psapi.lib, shlwapi.lib	adpcore.lib, adpcppf.lib, psapi.lib, shlwapi.lib

The SDK uses static linking to ensure that multiple versions of the libraries can be used simultaneously. An application and component must be compiled for the same API Level. See the API Versions and Levels section for details about versioning.

Debug and Test

Debugging and testing an Intel® Atom™ Developer Program applications and components is different than the way you would a normally debug applications due to the way the licensing system uses IDs.

Since a Debug ID cannot be used with the production Store Client, the SDK comes with a utility, the Application Test and Debug Service (ATDS), which allows you to run the application locally while it "emulates" the production infrastructure. This only works if the application has been compiled using a Debug ID. In fact, one way to test how well an application handles an **ADP_UNAUTHORIZED** error is to run it against the ATDS using a Production ID.

Debugging

Prior to running your application or component in the debugger you need to start up an instance of the ATDS.

1. Open a cmd window
2. Run the **runATDS.bat** file

The ATDS console will show up. You can now debug your application. The ATDS emulates the production client by providing your application with the correct API behavior (application ID excluded).

To stop the ATDS:

1. Open a cmd window
2. Run the **stopAtDS.bat** file

Testing

The SDK libraries/frameworks return specific return values or raise exceptions depending on the runtime to indicate errors. The ATDS can simulate these error conditions in response to various inputs in order to help with testing. See the Test section in the this document's appendices for error cases and conditions.

Submit the Application to the Store

Once you are satisfied with the application, it can be submitted to the Store for validation and distribution. In order to submit an application you must first obtain a Production application ID for your application by visiting "http://appdeveloper.intel.com", logging into your account, and a creating a new application. The system creates and displays a Production application ID which you should cut and paste into your application source code replacing the old Debug ID. Next, recompile your application and follow the directions on the site to upload the application.

At this point the application cannot be used or tested locally – it has a Production ID. It can only be downloaded and installed from the Store.

Writing an Application

This section describes, in detail, how to write an Intel® Atom™ Developer Program application.

License Enforcement

A machine is authorized to run an application if the application was purchased by the user on the machine or if it is one of the five systems allowed by the licensing system. As the developer of the application you are responsible for verifying that the current machine is authorized to run the application - the OS or runtime does not enforce this automatically.

C++ Framework

There are two basic approaches to creating an application:

- Add the framework - If you do not want to subclass the application class, you can integrate with C++ Framework by instantiating an application object and using it as a delegate to call the C++ Framework methods (see below).
- Define a new application by subclassing the application class. This is the recommended approach for new applications. It is the most elegant implementation and makes it easier to benefit from enhancements to the C++ Framework in the future.

> **NOTE:** Regardless of the method uses, the can be only one instance of an application or an application-derived class.

Delegate

This is a simple example of using the Delegate approach. In this case **pApp** is acting as the delegate.

```
#include "stdafx.h"
#include "adpcppf.h"
int _tmain(int argc, _TCHAR* argv[])
{
   Application *pApp = NULL;

   try {
       pApp = new Application(ApplicationId(ADP_DEBUG_APPLICATIONID));
   } catch (AdpException& e) {
      cout << "The attempt to authorize the application failed: "
            << e.what() << endl;
      if (pApp != NULL)
        delete pApp;
```

```
    }
```

You application code code goes here.

When the application exits remember to delete the delegate instance. Any components used by the application have to be deleted individually.

```
      if (pApp != NULL)
            delete pApp;
      return 0;
}
```

Sub-classing Application

See the "Putting it All Together" section below for a complete example.

The constructors for both component and application throws exception if there is an error, or if the product is not authorized for use. In C++, objects that have exceptions thrown in their constructor are not valid for use.

In order to prevent unnecessary overhead, as well as enforce proper application design constraints, the following behavior has been provided.

Application::Application

The application class stores static state variables, which are initialized with the first instance to be allocated in the process scope. The initialization process includes initializing the core libraries, authorizing the application for use by the current user, and future releases, and checking for product updates. One and only one instance of the application is allowed. If additional attempts to allocate an instance are made, the constructor throws an exception. The constructor calls are thread-safe.

It is important to note that in the event of an exception during its constructor, an application object is invalid for later use. This means that any logic, including UI functionality, is not accessible. It is recommended that developers creating applications built upon a graphic interface should code for such conditions, and provide UI capable logic for failure mode handling outside the scope of the application object.

C Language API

C Language API based applications are very simple: you initialize the library and authorize the machine. That's it.

Applications are identified by an application ID, either a production Id received from the Developer Portal or a Debug ID during development. You need one to build an application. It should be declared const to avoid any inadvertent modification. application IDs cannot change while an application is running. In fact only the first one passed to **ADP_IsAuthorized()** is used.

Below, we walk through a C Language API –based application.

```
#include <stdio.h>
#include "adpcore.h"
int main( int argc, char* argv[] )
{
   ADP_RET_CODE ret_code;
   const ADP_APPLICATIONID myApplicationID = {{
     0x12345678, 0x11112222, 0x33331234, 0x567890ab}};
. . .
```

After we define the ID, the next step is to initialize the runtime and check for success

```
. . .
```

```
   if ((ret_code = ADP_Initialize()) != ADP_SUCCESS ){
      switch( ret_code ) {
        case ADP_INCOMPATIBLE_VERSION:
           printf( "ERROR: Incompatible version" );
           break;

        case ADP_NOT_AVAILABLE:
           printf("ERROR: Client Agent not running" );
           break;

        case ADP_FAILURE:
        default:
           printf("ERROR: Unknown error" );
           break;
      } // switch

      exit( -1 );
    } // if-then
. . .
```

If **ADP_Initialize()** does not return **ADP_SUCCESS** there is very little you can do other that notify the user and exit. Notice the use of the **ADP_Close()** function. You should call this function whenever your application exits to ensure an orderly teardown of the runtime.

Next, you need to check to see if the current machine has a valid license to run you application by calling **ADP_IsAuthorized()** and passing it your application ID.

```
. . .
if (( ret_code = ADP_IsAuthorized( myApplicationId ))
        != ADP_AUTHORIZED ) {
      // handle error conditions
                . . .
   }

   if ( ret_code == ADP_NOT_AUTHORIZED ) {
      printf( "ERROR: this application is not authorized to run" );
      ADP_Close();
      exit(-1);
   }
. . .
```

ADP_IsAuthorized() returns two types of return values: error conditions and authorization statuses. (Error conditions are ignored as in the example above – see the C Language API Reference for details.).

There are three authorization statuses: Authorized, Not Authorized and Authorization Expired. The meaning of the first two is obvious but the last bears some explaining. Application licenses are cached locally and are periodically refreshed from the back-end. This allows applications and components to work while offline. If the machine does not connect back to the backend over a long interval to refresh its license a **ADP_AUTHORIZATION_EXPIRED** return code may occur. The solution is to have the user connect back to the network to refresh the license.

Once an application has been successfully authorized, you can call other functions in the C Language API (but not before). This would be a good time to trigger an application Instrumentation event like Begin to record the usage of the application.

```
. . .
  if (( ret_code = ADP_ApplicationBeginEvent())!= ADP_SUCCESS) {
      // handle error conditions
  }
                    . . .
```

At this point your core application logic should begin – there is no further interaction needed with the C Language API unless you use the Crash Reporting capability of the API (see section below for details.)

When your application exits it should call **ADP_Close()** to gracefully tear down the runtime, and you may want to trigger the End Application Instrumentation event to record the duration the application was used.

```
. . .
  if (( ret_code = ADP_ApplicationEndEvent())!= ADP_SUCCESS) {
      // handle error conditions
  }
  ADP_Close();
  exit(0);
)
```

Writing a Component

This section describes, in detail, how to write an Intel® Atom™ Developer Program component.

License Enforcement

An application is licensed to use a component if there is a pre-arranged agreement between the developer of the component and the developer of the application (see the Intel® Atom™ Developer Program Web site for more details). It is the responsibility of you, the component developer, to enforce the license, i.e. ensure that the calling application has a license to use the component.

C++ Framework

It is strongly recommended that components be developed using inheritance from the C++ Framework component class, though it is possible to write components using the delegation approach. C++ also allows for multiple inheritance, which could ease the repurposing of existing code for use as a component.

The constructors for both component and application throw an exception if there is an error, or if the product is not authorized for use. In C++, objects that have exceptions thrown in their constructor are not valid for use.

In order to prevent unnecessary overhead, as well as enforce proper application design constraints, the following behavior has been implemented.

Component::Component

While multiple instances of a component are allowed, the authorization logic is only executed upon the first instance allocation. The initialization process includes authorizing the component for use with the current application, as well as verification that the core libraries

are functional. Developer of an application must create an instance of application class before he/she could create an instance of the component.

Note that you may want to provide a single component package that provides multiple related classes for use by applications. In that case, rather than packaging them separately and having separate component Id's, a class factory approach is recommended. This allows a single component instantiation and authorization to provide an arbitrarily large number of different classes for use by subscribing applications.

C Language API

If you want to distribute a library of C functions as a component you need to do the following things:

- Inside your library you should define a variable to hold the authorization status of the application. You may want keep that variable hidden and avoid exporting it through the interface of your component.
- You should provide your own authorization function that encapsulates the **ADP_IsAppAuthorized( )** call. If you would like to make sure that an application cannot use your component until it has successfully authorized itself, you should check the variable holding the application's authorization status in all functions exported by your library.
- In your component's documentation you should provide the application developer with the information what function to call to authorize the application to use the component.

Putting It All Together

This section shows how applications can be combined with components

C++ Framework

In this sample, an application class, **SampleApplication**, and a component, **GoodiesComponent,** are used together.

The application developer decided to derive his application, **SampleApplication,** from the C++ Framework **Application** class.

```
class SampleApplication: public Application {
private:
      int data;
      static ApplicationId id;
public:
      SampleApplication(int data);
      static void PrintAuthorizationStatus();
      int GetData();
};
```

```
ApplicationId SampleApplication::id =
      ApplicationId(ADP_DEBUG_APPLICATIONID);

SampleApplication::SampleApplication(int data) : Application(id) {
  this->data = data;
};
```

```
void SampleApplication::PrintAuthorizationStatus() {
   cout << "Authorization status for " << (const char*) id << ": " <<
            Application::GetAuthorizationStatus() << endl;
};

int SampleApplication::GetData() {
  return data;
}
```

The component developer created a component, **GoodiesComponent**, deriving from the framework **Component** class.

```
class GoodiesComponent: public Component {
private:
      int Goodies;
      static ComponentId id;
public:
      GoodiesComponent();
      static void SayHello();
      int GetGoodies();
};
```

Implementation

```
ComponentId GoodiesComponent::id(ADP_DEBUG_COMPONENTID);

void GoodiesComponent::SayHello() {
      cout << "Hello" << endl;
};

int SampleComponent::GetGoodies() {
      return Goodies;
};
```

Here’s the code for the application:

```
#include "stdafx.h"
#include "SampleApplication.h"
#include "GoodiesComponent.h"
#include "InitializationException.h"
#include <iostream>

using namespace com::intel::adp;

int main(int argc, char** argv) {
. . .
```

Clear the application and component objects.

```
   SampleApplication* pSampleApplication = NULL;
   GoodiesComponent*  pGoodiesComponent = NULL;
```

Create instances of the application and the component within a try/catch block. If there are any exceptions such as Authorization, the control goes to the catch block

```
. . .
try { // Catch exceptions
      pSampleApplication = new SampleApplication(2);
      pGoodiesComponent = new GoodiesComponent();
   } catch (UnauthorizedException& e) {
      cout << "Caught UnauthorizedException in application: " <<
          e.what() << endl;
   }
. . .
```

At this point the machine is authorized to run the application and the application is authorized to use the component. Now developer may attempt to invoke a method of the component.

```
. . .
   if (pGoodiesComponent) {
      // a simple call to a Component method
      pGoodiesComponent->SayHello();
   }
. . .
```

And invoke a method of the application.

```
. . .
    if (pSampleApplication) {
      pSampleApplication->PrintAuthorizationStatus();
   }
. . .
```

Clean up by deleting the application. This also causes the component destructors to be called.

```
. . .
  delete pSampleApplication;
  // this will cause the destructors to be invoked
  // on the Sample Application.  Components will need to be
  // deleted independently.
  return 1;
}
```

C Language API

In this sample an application incorporates a third party component, **goodies.lib** (a C Library).

The component developer created a library that exports

```
ADP_RET_CODE SayHello();
ADP_RET_CODE AuthorizeMe( void );
```

And codes the library as follows:

```
. . .
bool ApplicationAuthorized = FALSE;

ADP_RET_CODE AuthorizeMe() {
```

```
   if (ret_code = ADP_IsAppAuthorized( ComponentId )) == ADP_AUTHORIZED
)
      ApplicationAuthorized = TRUE
};

int SayHello() {
  if ( ApplicationAuthorized )
    printf("Hello");
};
. . .
```

To use the "component" functions, the application developer needs to first call the **AuthorizeMe()** function after they've already authorized themselves using the **IsAuthorized()** function. If they are authorized to run on the machine and authorized to use the component it returns **ADP_AUTHORIZED**. Any attempt to use the **SayHello()** function without a successful authorization fails.

API Versions and Levels

The SDK libraries have a mechanism for managing versioning between the API/interfaces of applications and components, and Client Agent. There are two constants which provide information about the version of the application Library. The API Version constant identifies the version of the API and the API Level constant identifies the relative version of the API.

C Language API

```
const wchar_t* ADP_API_VERSION;
const unsigned long ADP_API_LEVEL;
```

C++ Framework

```
namespace com::intel::adp

const wchar_t* ADP_API_VERSION;
const unsigned long ADP_API_LEVEL;
```

The API Version constant is a human readable string of the API version, ie. "1.23.344-beta" and is not used for testing compatibility. Instead, it is the API Level constant that is used to determine compatibility between the application libraries and the Client Agent. Although, the API Level of an Application Services Library must be the same as the Client Agent, a Client Agent may support a range of API Levels concurrently. For example, there could be several releases of the application libraries, each with a different **ADP_API_VERSION** value that shares the same **ADP_API_LEVEL**.

If the APIs level of the application Library and the Client Agent are incompatible, **ADP_Initialize()** function returns **ADP_INCOMPATIBLE_VERSION** in C. In C++ the framework raises an exception.

> **NOTE:** Applications should notify the user of the issue and allow he or she to resolve it via the Store Client. An application should not attempt to upgrade itself.

An application and its components must be linked with a library of the same **API_LEVEL**, or else the application fails to initialize.

Crash Reporting

Best practices for application design include the trapping of all exceptions in code. If an error occurs that is unrecoverable, you should always try to notify the end user and exit gracefully. In the event that this is not possible, or not done by the developer, an application crashes.

Your dashboard at the Intel® Atom™ Developer Program site can provide information about crashes your applications may have suffered. You use the Crash Report functions to create the crash reports.

When a crash report is created, it is first written to disk and then a wake-up message is sent to a process responsible for sending the report to the Developer Portal. The process also periodically sweeps the crash report area looking for unprocessed crash reports

The crash report stores locally on disk and is a subset of the final report. See the Crash Report section in the Appendices for an example crash report and a list and description of the data it contains when it reaches the Developer Portal.

> **NOTE:** There is no guarantee that a Crash Report will be generated since the stability of the system cannot be assumed. The library has been designed to take the "shortest path" to disk to maximize success. You should apply this rubric as well – keep the use of custom report fields and complex logic to the bare minimum.
>
> Crash reports are only to be made in the event of an application crash, and are not to be used as a logging solution. Abuses will be monitored.

The SDK automatically populates several fields of the crash report. You can (optionally) provide values for other predefined fields in the report (see the Developer References for a list and description). Additionally, crash reports have two areas where you can insert your own debug/troubleshooting information: **ErrorData** and custom report fields. **ErrorData** is a good place to attach a stack trace or a long error message while custom report fields allow you to add up to 20 name/value pairs to a crash report.

> **NOTE:** You are responsible for ensuring that Crash Reports do not contain any data that violates Intel Privacy rules (see "http://appdeveloper.intel.com" for details)

The **Category** crash report field has special significance: the Developer Portal allows you to group crash reports using this field. To help simplify troubleshooting and reduce the amount of data you see, you may want to create a classification scheme, e.g. MEMORY, IO, USER ERROR, and so on to use with **Category**.

For a list and description of the all data fields in a Crash Report and an example Crash Report see the Crash Reporting section in the Appendices.

C Language API

When using the C Language interface, you are responsible for catching errors or faults and calling **ADP_CrashReport()** yourself. To catch error conditions, it is recommended that Signal Handler are utilized. Signals are messages sent to the C runtime, which then executes callbacks registered to handle these messages. In the Microsoft* C runtime (CRT), signals are documented in the following link:

"http://msdn.microsoft.com/en-us/library/xdkz3x12%28VS.71%29.aspx"

No integrated error handling/crash reporting is provided (use the C++ Framework for an integrated solution).

C Language API

```
. . .
   // Setup sample crash handler for SIGABRT signal
   if(signal(SIGABRT, SampleCrashHandler)==SIG_ERR)
   {
      printf("An error occured while setting a signal handler.\n");
      exit(0);
   }
. . .
```

```
// Sample crash handler
void SampleCrashHandler(int signal)
{
      // API return code
      ADP_RET_CODE response;

      WCHAR *module = L"AdvancedApp";
      unsigned int lineNumber = __LINE__;
      WCHAR *message = L"Aborted the operation";
      WCHAR *category = L"Critical";
      WCHAR *errorData = L"Invalid Parameter";
      unsigned long errorDataSize = wcslen(errorData);
      errorDataSize = wcslen(errorData);
      response = ADP_ReportCrash(module,
                           lineNumber,
                           message,
                           category,
                           errorData,
                           errorDataSize,
                           NULL,
                           0);

      if( response != ADP_SUCCESS )
      {
            printf("FAIL: ADP_ReportCrash failed with error code%d\n",
                  response);
      }
}
```

Customizing the Crash Report

A Crash Report is extended by declaring an array of **ADP_CrashReportField**, setting the name and values, and passing it to **ADP_ReportCrash()**.

C Language API

```
{
   . . .
   ADP_RET_CODE response;
   // Custom Crash report fields
   ADP_CrashReportField crashReportField[ CUSTOM_FIELD_NUMBER ];
   crashReportField[0].name= L"Device";
   crashReportField[0].value= L"Netbook";
   crashReportField[1].name= L"OS";
   crashReportField[1].value =L"Win7";
   response = ADP_ReportCrash(L"MyApplication",
```

```
            __LINE__,
            L"Unknown Error",
            L"Critical" ,
            L"Error Data",
            strlen("Error Data"),
            crashReportField,
            CUSTOM_FIELD_NUMBER);
   exit( -1 ); // exit with an error
}
```

Remember the less you try to do when handling the crash, the greater the chance of the Crash Report making it to disk.

C++ Language Framework

In the event of a crash, the C++ runtime provides a termination handler (or callback) that is leveraged by the SDK framework. The framework processes the exception in the callback, and provides you with whatever information is available.

To aid in gathering information about the cause, the ADP C++ framework provides a class, CallStack, that traverses the stacks of both Microsoft Windows* and Linux* environments, and collects stack frame information. In Microsoft Windows* environments, this code leverages the DBGHELP.DLL library to provide symbol table, module, and line number information. It is important to note that there are many different versions of DBGHELP over the install base for ADP applications. Because of this, two modes of CallStack have been released with the SDK. In the default, basic, mode, the CallStack class logic traverses the main application thread's callstack by address only. However, if more detailed stack trace information is desired, the following steps are necessary:

1. Download and install DBGHELP library, available with the Microsoft Debugging Tools for Windows* package
2. Define _ADP_ADVANCED_CRASHREPORT pre-processor symbol in the project file and build IADP SDK application
3. Redistribute DBGHELP.DLL with the application in the application's root directory to ensure that the correct version of the DLL is available at runtime on the end user's system. Refer to the DBGHELP licensing documentation for terms and conditions of its redistribution

In Linux*, the backtrace API set is used. For Linux* applications, you must link your application with the flag -rdynamic, which provides the greatest amount of information to backtrace. Without this setting, only module and address offsets are available.

In the case that the your code has caught an exception from which the application cannot continue, you will have access to **Product::pClientProxy**, which is a wrapper of the SDK C library functions (see the C Language Developer Reference for details). Through the **ClientProxy**, you can submit a crash report. For example:

```
. . .
      try {
            IllegalMethod();
      } catch (exception& e) {
            printf("Caught an exception: %s\n", e.what());

            // We decide that we can't continue, create a crash report
            wchar_t* moduleName = L"TestApplication.cpp";
```

```
            wchar_t* message = L"Caught Exception!";
            wchar_t* category = L"System Exception";
            ADP_CrashReportField fields[2];
            long lineNumber = __LINE__;

            fields[0].name = L"Sub Category";
            fields[0].value = L"Trapped Error";
            fields[1].name = L"Application Mode";
            fields[1].value = L"Initialization";

            pClientProxy->ReportCrash(moduleName, message, category,
                  lineNumber, NULL, fields, 2);
            exit(-1);
      }
}
```

Customizing Crash Reporting

You can customize crash reports using the following strategies:

1. For caught exceptions, you can may call directly into **Product::ClientProxy::ReportCrash** (see above). Additionally, you can instantiate a CallStack object, invoke CallStack::Parse, and provide that stack information as additional text for errorData. See the C++ Framework Specification for more information, or the header CallStack.h in the ADP C++ SDK.
2. For uncaught exceptions, you may subclass either **AbstractCrashReport** or **DefaultCrashReport** and provide an instance of this class to the application via the **Application::SetCrashReport** method.

Example:

Create a custom Crash Report derived from **DefaultCrashReport**

```
class ExampleCrashReport: public DefaultCrashReport {
protected:
      virtual void PopulateCrashReportFields();
};
```

```
void ExampleCrashReport::PopulateCrashReportFields() {
      CallStackEntryList entries = callStack.GetEntries();

      // Look at the top entry, perhaps that can tell us more
      // about the cause of the exception, which we can then
      // indicate in report fields.
      CallStackEntry entry = entries.pop_front();
      if ([some conditional based on CallStackEntry]) {
            pCrashReportFields = new ADP_CrashReportField[2];
            pCrashReportFields[0].name = L"Sub Category";
            pCrashReportFields[0].value = L"Trapped Error";
            pCrashReportFields[1].name = L"Application Mode";
            pCrashReportFields[1].value = L"Initialization";
            reportFieldCount = 2;
```

```
        }
}
```

In the case of uncaught exceptions, the application is initialized with an instance of **DefaultCrashReport** to dispatch uncaught exceptions leading to application faults.

```
Application::SetCrashReport(new ExampleCrashReport);
```

AbstractCrashReport

AbstractCrashReport is an abstract class with pure virtual methods that are designed to populate data existing within the class. Since the system may be at an unstable state, allocation of data on the stack is to be avoided, if possible. These methods will be called by the underlying framework to collect data to be submitted in crash reports. By implementing a subclass of **AbstractCrashReport**, you can gain control of every field that will be returned during the crash reporting process. However, you are also required to implement every field.

DefaultCrashReport

The **DefaultCrashReport** class is a reference implementation of the **AbstractCrashReport** class and is installed by default. This default implementation performs best-effort collection of data that should be appropriate and accurate in most cases. However, you may subclass **DefaultCrashReport** to customize one or more fields (see the C++ Framework Developer Reference for more information)

AdvancedCrashReport (Microsoft Windows* only)

The **AdvancedCrashReport** class provides an advanced implementation of the **AbstractCrashReport** class. This implementation leverages a class **AdvancedCallStack**, which inherits from **CallStack**, and provides full callstack presentation (module, symbol, line number). To provide these enanced capabilities **AdvancedCrashReport** class utilizes **DbgHelp** utility library distributed as part of the Microsoft Debugging Tools for Windows* package. For more information on how to use **AdvancedCrashReport** with your application refer to the SDK C++ Framework Reference document.

Application Instrumentation

Your dashboard at the Intel® Atom™ Developer Program site can provide application usage statistics. You use the application Instrumentation functions to send these events to the Developer Portal.

The application events are triggered within the application and are sent to the Client Agent which forwards them to the back end for aggregation, or if offline, caches them for later transmission.

> **NOTE:** The final decision on whether instrumentation events are returned to the Developer Portal sits with the end user. The user must agree to the collection and transmittal of the information. If he or she says no, the data is discarded.
>
> Also**,** the data is aggregated in the Developer portal and no user or machine identifiable data is available

The APIs can be used to record the number of times an application is run and the duration of the run can be recorded.

C Language API

```
#include "adpcore.h"
int main( int argc, char* argv[] )
{
   // Initialize and Authorized Application;
```

```
                . . .
   // Record Application start
   ret_code = ADP_ApplicationBeginEvent(  );
   //Core application code
                . . .
   // Record Application end
   ret_code = ADP_ApplicationEndEvent(  );
   exit(0); // Application exit
}
```

C++ Framework

```
#include "adpcppf.h"
#include <iostream>

int main( int argc, char** argv ) {
   Application *myApp = new Application( ApplicationId::DEBUG_ID );
   try {
      myApp->BeginEvent();
   } catch (AdpException& e) {
      cout << "An exception occured invoking BeginEvent: " <<
         e.what() << endl;
      exit();
   }
. . .
   try {
      myApp->EndEvent();
   } catch (AdpException& e) {
      cout << "An exception occured invoking EndEvent: " <<
         e.what() << endl;
      exit();
   }
}
```

If you want to track only the number of times an application is used you need only call **BeginEvent**. To track the duration of use, you need to call both **BeginEvent** and **EndEvent**. Call to the **EndEvent** without previous call to the **BeginEvent** will result in **ADP_NO_APP_BEGIN_EVENT** exception..

Application Instrumentation APIs are not available to component developers.

Error handling

The C Language API uses a set of predefined return values and the Framework uses a combination of return values and exceptions to communicate the success or failure of a call.

In the case of the C Language API, you should always check the return value of each function call; especially given that some calls rely on the services that may become unavailable in the course of an application's execution. In the case of C++, all calls into the framework should be wrapped in try/catch block*s*. Even if you're expecting a simple result, it is possible that the

operating environment will have changed since the last call and that an exception could be thrown.

For instance, you might have successfully instantiated an application object without receiving an exception. But, upon calling **Application::BeginEvent**, you may receive an exception because the Client Agent service had been stopped by the user in the meantime.

C Language API

Return Codes

The return codes for the C Language API are defined as manifest constants in the header file. All functions share common return code for common conditions and some error condition are recoverable. **ADP_FAILURE** is returned when the library encounters an unknown error. Please see the C Lang2uage API Developer Reference from more details

C++ Framework

Exception Classes

All of the C++ Framework exceptions are derived from the base **AdpException** class, which in turn is based on the exception class supported by the runtime. These exceptions include: **InitializationException**, **AdpRuntimeException**, and **UnauthorizedException**. The base class provides three key pieces of information which will be available in all specialized subclasses, even if they add additional information.

- **code()** provides a numeric value that will usually correspond to one of the return codes, such as **ADP_NOT_AUTHORIZED** or **ADP_FAILURE.**
- **message()** provides a string description of the error suitable for display to end users, such as "The application could not be authorized."
- **what()** provides a concatenation of **code()** and **message()** in the form of "The application could not be authorized (Code: 23)".
- **Overloaded operator <<()** provides the developer the ability to use AdpExceptions with cascading << operators with both ostream and wostream (cout and wcout) objects. The operator is equivalent to what().

You can trap all possible exceptions thrown by the framework by implementing exception handling as follows:

```
try {
     [Invoke some C++ Framework method or constructor]
} catch (AdpException& e) {
     [Notify user of error, terminate gracefully]
}
```

Appendices

Crash Reporting

Fields

This table lists the data available in in final Crash, i.e. the one that is sent to the back end, and who is responsible for providing the data.

Crash Report Field	Source
Application Name	SDK
Application ID	SDK
ModuleName	Developer
LineNumber	Developer
Category	Developer
Message	Developer
ErrorData	Developer
SDK API Level	SDK
Runtime	SDK
Runtime version	SDK
OS	SDK
OS version	SDK
Timestamp	SDK
Up to 20 custom fields	Developer
Store Client Version	Store Client
Agent ID (HL client ID)	Store Client

Example: Crash Report

```
<?xml version="1.0" encoding="UTF-8" ?>
<CrashReport>                    <!—Hardcoded Crash report header -->
    <DummyApp>          <!—Application name (process name) -->
        <!—Name of the module where the crash happened. Provided by
            Application -->
         <ModuleName>DummyApp.c</ModuleName>
        <!—Line number of the module where the crash happened. Provided
            by Application -->
         <LineNumber>100</LineNumber>
        <!—Brief error description. Provided by Application -->
```

```
        <Error>Unknown Error</Error>
       <!—Crash category used to bin crash reports. Provided by
           Application -->
        <Category>Critical</Category>
       <!—Managed runtime. Filled in by SDK core library -->
        <Runtime>Not implemented</Runtime>
       <!—Managed runtime version. Filled in by SDK core library -->
        <RuntimeVersion>Not implemented</RuntimeVersion>
       <!—OS name. Filled in by SDK core library -->
        <OS>Windows 7 Ultimate</OS>
       <!—OS version. Filled in by SDK core library -->
        <OSVersion>6.1</OSVersion>
       <!—IADP SDK version. Filled in by SDK core library -->
        <SDKVersion>0.9</SDKVersion>
       <!—Date and time of crash. Filled in by SDK core library -->
        <Timestamp>2009-11-20T22:25:27.000</Timestamp>
       <!—Upto 4000 characters long data field attached to the
           crash.Provided by Application -->
        <ErrorData>Error Data</ErrorData>
       <!—Upto 20 user defined crash report fields.Provided by
           Application -->
       <customfields>
           <!—Name – value pairs -->
            <customfield name="moon">full</customfield>
            <customfield name="wind">nne</customfield>
        </customfields>
    </DummyApp>
</CrashReport>
```

Test Case Matrix

This table list error conditions and how to cause then when using the ATDS.

Error Condition	Application Action	Application Test Debug Service
Authorized Application	Call IsAuthorized with Application ID = DEBUG_ID	Return a response with CreationDate = today ExpirationDate = today + 10 days
Unauthorized Application	Call IsAuthorized() with an Application ID != DEBUG_ID	Return a response with CreationDate = in the past ExpirationDate = in the past
Authorization Expired	Call IsAuthorized() with Application ID = DEBUG_ID	TBD
Component Application	Call IsAuthorised() with Application ID =	Return a response with

Authorization	DEBUG_ID Call Is AppAuthorized() with Component ID = DEBUG_ID	CreationDate = today ExpirationDate = today + 10 days
Unauthorized Component	Call IsAuthorized() with Application ID = DEBUG_ID Call Is AppAuthorized() with Component ID != DEBUG_ID	Return a Application response with CreationDate = today ExpirationDate = today + 10 days Return a Component response with CreationDate = in the past ExpirationDate = in the past
Incompatible version	Call **Intialized()**	Start the ATDS with a flag -i. Behaves as if it's API_LEVEL is less than the Application Library
Not initialized	Call any function without a call to **Intialize()**	---
Not available	Start up application with an instance of the ATDS running	Don't run the ATDS
NO Application Begin Event	Call **ApplicationEndEvent** without a previous call to **ApplicationBeginEvent**	--