

Intel® Atom™ Developer Program SDK C++ Framework Reference

November 2009

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.

Introduction

This document is the reference for the Intel® Atom™ Developer Program SDK C++ Framework. It covers API_LEVEL 1 and API_VERSION 0.9.

Background

The Intel® Atom™ Developer Program SDK C++ Framework (C++ Framework) provides a high level C++ interface to the Intel® Atom™ Developer Program's underlying C implementation. The C++ Framework does this by organizing the APIs into an object model that takes advantage of C++ features to reduce complexity and to provide for easy future expansion.

Working with the C++ Framework

In conjunction with the Intel® Atom™ Developer Program SDK, the C++ Framework provides a means for developers to write, test, and deploy applications for the Intel® Atom™ Developer Program.

Writing Applications and Components

The C++ Framework can be used as part of a new project or added to an existing project. Note that some functions implemented in the underlying C library are absent in C++ Framework because they have been incorporated into the framework.

Assumptions

In many cases, there is clear default behavior that most applications and components will want to follow. For instance, if an application's **GetAuthorizationStatus** call returns **ADP_NOT_AUTHORIZED** the normal course of action would be to notify the user of the problem and exit the application. However, the framework does not enforce this behavior, and it is up to the developer to determine how to proceed (e.g., operate in demo mode).

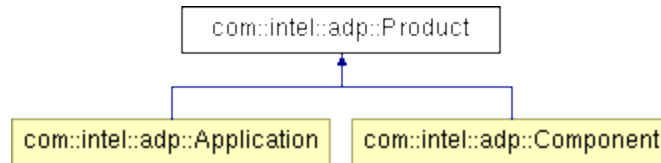
Class Structure

The C++ Framework is broken down into three hierarchies based on the following classes: **Product**, **AbstractCrashReport**, and **AdpException**. All application and component core functionality is incorporated in **Product**, and it is not necessary for developers to implement the crash reporting interfaces; a default implementation is provided that catches unhandled exceptions and reports them to the back end on the developer's behalf. However, extending or overriding the crash reporting classes may provide additional value to developers, either by enhancing the data in crash reports or by replacing the crash report mechanism entirely. All exceptions thrown by the C++ runtime are based on **AdpException**, providing developers with the option of catching all exceptions in a single catch filter, or expanding to multiple catch expressions for conditional error handling. **AbstractCrashReport** is provided for uncaught exception handling, which will be discussed in a following section in this document.

Class Reference

All classes are define within the **com::intel::adp** namespace.

Product



The **Product** class is the root of the C++ Framework’s class hierarchy. The class, in addition to establishing static utility methods and abstract overrides, provides a static, thread-safe map that maintains the relationship between developer class types and framework Product entries.

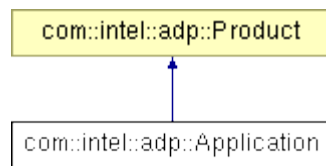
A product can be:

- An application
- A component

Public Member Functions

| | |
|----------------|-------------|
| | Product () |
| virtual | ~Product () |

Application



Developers extend the **Application** class to create Applications for the Intel® Atom™ Developer Program.

Only one instance of the **Application** class can be created. The Application constructor makes calls to the C Library functions **ADP_Initialize()** and **ADP_IsAuthorized** and will throw exceptions if they fail. In this case the instance of the **Application** class will not be constructed and its further use may lead to undefined behavior. Developers are advised to wrap Application’s creation in a try-catch block, and handle constructor’s exceptions appropriately.

While developing an application, the **Application** constructor should be called with *ApplicationId* set to **ADP_DEBUG_APPLICATIONID** . Before submitting the application to the

Intel® Atom™ Developer Program, a valid Application ID should be obtained from the Intel® Atom Developer Portal and used for *ApplicationId*. See the Developer Guide for more details.

Summary

| Public Member Functions | |
|-------------------------------|--|
| | Application () |
| | Application (ApplicationId id) |
| virtual | ~Application () |
| static void | TerminateHandler () |
| static void | SetCrashReporter (AbstractCrashReport *pCrashReporter) |
| static ADP_RET_CODE | GetAuthorizationStatus() |
| void | BeginEvent() |
| void | EndEvent() |

Application (ApplicationId id)

A constructor for **Application** that sets up the framework and underlying libraries. The constructor will throw exceptions if the Application object is instantiated more than once or if there is a problem authorizing the application. In the event of an exception during a constructor call, the object instance will be invalid and non-static methods on the object will be unavailable.

If **AdpRuntimeException** or **UnauthorizedException** is thrown, developers should call the static method **Application::GetAuthorizationStatus()** to determine which specific problem occurred and what action to take.

During initial development of an application developer can pass **ADP_DEBUG_APPLICATIONID** to the **Application** constructor. However, a valid **ApplicationId** must be used when submitting the application to the Developer Portal. See the Developer Guide for more details.

Note that in the event of exception during construction of the Application object, the object will not be valid for the later use. Additionally, if developer inherits from the Application class, the constructor of the derived class will not be executed if Application constructor throws. It is recommended that developers provide logic for handling Application constructor failures outside of the scope of the Application object.

Parameters

| Name | Description |
|------|-------------|
|------|-------------|

| | |
|-----------|--|
| <i>Id</i> | The ApplicationId of the application. |
|-----------|--|

Exceptions

| Name | Description |
|--------------------------------|--|
| AdpRuntimeException | Thrown if a critical error occurs in the Intel® Atom™ Developer Program core library |
| InitializationException | Thrown if the application has already been instantiated |
| UnauthorizedException | Thrown if there is a problem authorizing the app |

~Application ()

This is the default virtual destructor, and will be automatically called by the runtime upon application object deletion.

Parameters

(none)

Exceptions

(None)

Returns

(None)

static **ADP_RET_CODE GetAuthorizationStatus()**

Returns the value of the authorization check that was performed by the constructor. Developers can assume that an exception-free constructor call is an indication of an Authorized status.

Parameters

(none)

Exceptions

(None)

Returns

| Value | Description |
|---------------------------|--------------------------------------|
| ADP_AUTHORIZED | Application is authorized to run |
| ADP_NOT_AUTHORIZED | Application is not authorized to run |

| |
|--|
| ADP_AUTHORIZATION_EXPIRED Application's license has expired |
|--|

static void **SetCrashReporter**(**AbstractCrashReport** **CrashReporter*)

Allows the developer to replace the means by which application crashes are reported to the Intel® Atom™ Developer Program back end. See Crash Reporting and the Developer Guide for more details.

Parameters

| Name | Description |
|----------------------|---|
| <i>CrashReporter</i> | A class derived from AbstractCrashReport |

Exceptions

(None)

Returns

(None)

static void **TerminateHandler** ()

The **TerminateHandler** will invoke **ReportCrash** on the currently installed instance subclassing **AbstractCrashReport**, as well as performing general cleanup in an effort to make a graceful landing from an unhandled exception. The **DefaultCrashReport** is provided in the **Application** by default. Developers, however, are free to provide their own subclasses of either **AbstractCrashReport** or **DefaultCrashReport**, and install the new version using **Application::SetCrashReport**. Please see the entries for these classes for more information.

Parameters

(None)

Exceptions

(None)

Returns

(None)

void **BeginEvent** ()

A call to this method records that the application was run. A subsequent call to the **EndEvent()** method must be made to record the duration of use.

Parameters

(none)

Exceptions

(None)

Returns

| Value | Description |
|------------------------------|--------------------------------------|
| AUTHORIZED | Application is authorized to run |
| NOT_AUTHORIZED | Application is not authorized to run |
| AUTHORIZATION_EXPIRED | Application's license has expired |

void EndEvent ()

A call to this method informs instrumentation framework that the application run has ended.

Parameters

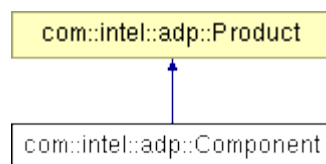
(none)

Exceptions

(None)

Returns

| Value | Description |
|------------------------------|--------------------------------------|
| AUTHORIZED | Application is authorized to run |
| NOT_AUTHORIZED | Application is not authorized to run |
| AUTHORIZATION_EXPIRED | Application's license has expired |

Component

Developers extend the **Component** class to create components that can be distributed via the Intel® Atom™ Developer Program Store and used in Intel® Atom™ Developer Program applications.

Unlike Applications, a component may be instantiated any number of times. The first time a component is instantiated by an application, the C++ Framework will perform the required authorization checks.

If there is any problem with authorization, **UnauthorizedException** will be thrown, and the developer may determine the specific problem by calling **GetAuthorizationStatus()**. If the Application object has not been instantiated before the Component, an **InitializationException** will be thrown.

It is up to the component developer to check the results of authorization and to act accordingly. Because only the first instantiation for each component performs those checks, there is very little performance penalty associated with repeated testing of results, or subsequent instantiation of component objects.

While developing a component, the component Id should set to **ADP_DEBUG_COMPONENTID** when calling the Component constructor. Before submitting the component to the Intel® Atom™ Developer Program, however, a valid **ComponentId** must be obtained from the Intel® Atom™ Developer Program and used for id. See the Developer Guide for more details.

Public Member Functions

Component (ComponentId id)

virtual **~Component ()** throw ()

Static Public Member Functions

static **GetAuthorizationStatus()**
ADP_RET_CODE

Constructor Component (ComponentId id)

Constructor for Component. Sets up framework and underlying C library and authentication. Only the first instance of a specific component will perform authorization checks, though all instances have access to the results by calling the static **GetAuthorizationStatus()** method.

While developing a component, *id* should be set to **ADP_DEBUG_COMPONENTID**. However, a valid **ComponentId** obtained from the Developer Portal must be used when a component is submitted to the Developer Portal.

Parameters

| Name | Description |
|-----------|--|
| <i>id</i> | The ComponentId of this component |

Exceptions

| Name | Description |
|--------------------------------|--|
| AdpRuntimeException | Thrown if an error occurs in the Intel® Atom™ Developer Program core library |
| InitializationException | Thrown if an the Application object has not been initialized |
| UnauthorizedException | Thrown if there is a problem with authorization |

Returns

(none)

Destructor: ~Component ()

This is the default virtual destructor, and will be automatically invoked by the runtime when the component instance is deleted.

Parameters

(none)

Exceptions

(none)

Returns

(none)

ADP_RET_CODE GetAuthorizationStatus()

Returns the authorization status for this component running in the current application context. It is up to the component developer to determine the appropriate action to take in the event that the component is not authorized.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|----------------------------------|---|
| ADP_AUTHORIZED | The application is authorized to use this component |
| ADP_NOT_AUTHORIZED | The application is not authorized to use this component |
| ADP_AUTHORIZATION_EXPIRED | Application's license to use this component has expired |

Exceptions

AdpException

AdpException, derived from the CRT provided exception class, is the base class from which all Intel® Atom™ Developer Program C++ Framework exceptions are derived. It provides a standard way to get information that spans exception types, and derived classes may or may not extend its functionality.

The **AdpException** object provides member variables to record a string for the message of the exception, as well as a long to record the corresponding code of the exception. **AdpException** provides the following methods to return state information.

const char* what()

Overrides **exception::what**. Returns the message and code of the exception in string form.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|--------|-------------------------------------|
| char * | A (const char*) pointer to a string |

const char* message()

Returns the message of the exception.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|--------|-------------------------------------|
| char * | A (const char*) pointer to a string |

long code()

Returns the code of the exception.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|-------|----------------|
| long | Exception code |

AdpRuntimeException

Represents an error condition in the underlying core library, such as a failure to contact the client proxy or a communication error between the client proxy and the Client Agent.

long code() can return:

| Value | Description |
|----------------------|---|
| NOT_AVAILABLE | The Client Agent is not available |
| FAILURE | General failure in communication or library |

InitializationException

Represents an error during the initialization of an **Application** or **Component**. At present, the only condition that causes an **InitializationException** to be thrown is an attempt to instantiate more than one **Application** with the same **ApplicationId**.

long code() can return:

| Value | Description |
|----------------------|---|
| NOT_AVAILABLE | The Client Agent is not available |
| FAILURE | General failure in communication or library |

UnauthorizedException

Represents a failure to cleanly authorize an application or component. For applications, this may indicate deployment on an unauthorized machine or lack of internet connectivity. For

components, this may represent an attempt at instantiation by an unauthorized application, or a lack of internet connectivity.

long code() can return:

| Value | Description |
|------------------------------|---|
| NOT_AVAILABLE | The Client Agent is not available |
| FAILURE | General failure in communication or library |
| NOT_AUTHORIZED | Application is not authorized to run |
| AUTHORIZATION_EXPIRED | Authorization token could not be refreshed |

Crash Reporting

The purpose of a crash report handler is to gather as much relevant detailed data about the state of a current application thread that has encountered an unexpected exception and to pass that information to the C Language API's **ReportCrash()** function so that it can be communicated to the back-end, making it available to developers for troubleshooting in the event of unhandled exceptions that users may be experiencing. To process an unhandled exception, the framework registers a callback with the C++ runtime that is invoked. This callback method, in turn, invokes a method on a subclassed instance of **AbstractCrashReport** maintained by the **Application** object.

By default, the **DefaultCrashReport** class is assigned to handle crashes, but may be replaced by the application developer. To implement your own crash report class, you can either create a subclass that inherits from the **DefaultCrashReport** class and override selected methods in that class or you can implement a subclass that inherits from **AbstractCrashReport** and implement all methods in that class.

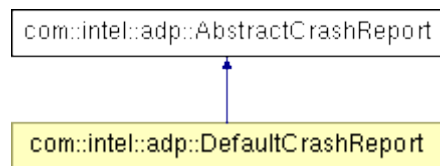
To set your custom crash report handler in your application, you call the **Application::SetCrashReporter** method, passing the new Crash Report handler object as a parameter. Once that is set, the new crash report handler will be used instead of the default crash report handler object. (NOTE: The crash report handler will only be invoked when an unhandled exception is encountered in the main thread in the current process space, but not when an unhandled exception occurs in a separate thread within the main process space.)

When following best practices, the application developer will catch an exception leading to application termination. In such cases, they are free to call into **Application::pClientProxy** to make the appropriate crash report with their own logic. The developer will have all the information needed to make accurate reports at their disposal.

In the event when an unhandled exception occurs, the application cannot and will not continue to execute. The crash report handler can only gather any relevant information (stack trace, static local and/or global variables or other relevant information that the current thread may have access to) so that it can be used to troubleshoot unhandled exceptions that occur in your application. Once the crash report handler has collected that information, it will attempt to pass that data to the ADP C Language API's **ReportCrash()** function. This call is not guaranteed to succeed and any attempt to do so can only be considered a best effort.

The context for the current crash handler is the main thread of the application and the **com::intel::adp** namespace. At the time the crash report handler is called, the state of the main thread of the application is by definition undefined, attempts to utilize the stack or even basic application state may not be possible.

AbstractCrashReport



AbstractCrashReport is the base class from which all crash reports are derived. You may inherit from **AbstractCrashReport** as long as you implement every method in the class; or, you may extend the **DefaultCrashReport** class if you only want to override some of these methods. The C method **ADP_ReportCrash** is developed against the following signature:

```

ADP_RET_CODE ADP_ReportCrash(
    const wchar_t      *Module,
    unsigned long      LineNumber,
    const wchar_t      *Message,
    const wchar_t      *Category,
    const wchar_t      *ErrorData,
    unsigned long      ErrorDataSize,
    ADP_CrashReportField *CustomFields,
    unsigned long      CustomFieldNumber
);
  
```

The arguments to this method have been declared in **AbstractCrashReport** as member variables with the following names:

```

wchar_t module[80];
wchar_t message[80];
wchar_t category[80];
wchar_t errorData[4000];
long lineNumber;
ADP_CrashReportField* pCrashReportFields;
long crashReportFieldCount;
  
```

Developers wishing to provide customized unhandled crash reporting will need to override the pure abstract methods in **AbstractCrashReport**, populating these variables. In turn, these variables will be used in the call to **ADP_ReportCrash**.

The **AbstractCrashReport** class offers the following virtual overrides

virtual void PopulateModuleName()

This method is provided to populate the **AbstractCrashReport::module** member variable.

virtual void PopulateLineNumber()

This method is provided to populate the **AbstractCrashReport::lineNumber** member variable.

virtual void PopulateMessage()

This method is provided to populate the **AbstractCrashReport::message** member variable

virtual void PopulateCategory()

This method is provided to populate the **AbstractCrashReport::category** member variable.

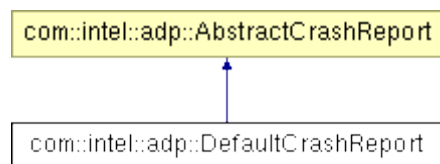
virtual void PopulateErrorData()

This method is provided to populate the **AbstractCrashReport::errorData** member variable.

virtual void PopulateCrashReportFields()

This method is provided to populate the **AbstractCrashReport::pCrashReportFields** and **AbstractCrashReport::crashReportFieldCount** member variables.

DefaultCrashReport



The **DefaultCrashReport** class provides a reference implementation of the **AbstractCrashReport** class. This implementation leverages a class **CallStack** which has provided for your use as an hpp file. In this configuration, all business logic is exposed as a header file, and all methods of the class are declared *inline*. **CallStack**, when invoked, will attempt to traverse the callstack of the calling thread, compiling information on each stack frame into **CallStackEntry** structures, and storing each into an STL templated list container.

The structure of a CallStackEntry is as follows:

```
struct CallStackEntry {
```

```

        wstring fileName;
        wstring moduleName;
        wstring symbolName;
        unsigned long lineNumber;
        void* pAddress;

        CallStackEntry();
        wstring str();
};

```

CallStack

In Microsoft Windows* environment, CallStack utilizes the Win32 API CONTEXT structure, along with pointer arithmetic, to traverse the stack and populate the CallStackEntryList with a set representing the stack frames of a given thread.

In Linux* based environments, the backtrace based APIs are used to provide stack frame information. As with the Microsoft Windows* version, the CallStack logic can provide either detailed debugging information, or only the module name and function address. The difference, however, is in how the developer links the application. If the linker flag `-rdynamic` is used, symbols and even line numbers will be available. If not, the data available will be restricted.

In either environment, CallStack features a no argument, default constructor, which initializes the object's internal variables. When it is desired for the CallStack class to record the state of a calling thread's stack, the developer will invoke the CallStack::Parse method. For Win32 environments, the CallStack::Parse method takes a CONTEXT structure. To retrieve this record arbitrarily, the developer can invoke the Win32 library method **::RtlCaptureContext**.

```

// Execute CallStack::Parse for Win32 at an arbitrary execution
// point.
CallStack callStack;
CONTEXT context;
::RtlCaptureContext(&context);

CallStackEntryList entries = callStack.Parse(&context);

```

In a Linux environment, the developer does not need to pass an argument to CallStack::Parse. The backtrace API provided by libc enables the retrieval of stack frame information without an explicit context structure, with the assumption that the stack trace is for the current calling thread. There is, however, one recommendation for the developer: To enable the Linux API with the ability to gather the most information possible about the stack, including symbol information and even line numbers in code, the application must be compiled with the linker flag `-rdynamic`. If this flag is omitted, only the module name and address offset will be available to the backtrace APIs.

CallStackEntryList& Parse()

Parses the calling thread's call stack, saving the information in a stl collection `list<CallStackEntry>`.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|--------------------------------------|---|
| <code>CallStackEntryList&</code> | A reference to the <code>CallStackEntryList</code> maintained by the <code>CallStack</code> object. |

CallStackEntryList& GetEntries()

Returns a reference to the `CallStackEntryList` maintained by the `CallStack` object.

Parameters

(none)

Exceptions

(none)

Returns

| Value | Description |
|--------------------------------------|---|
| <code>CallStackEntryList&</code> | A reference to the <code>CallStackEntryList</code> maintained by the <code>CallStack</code> object. |

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 enhanced capabilities **AdvancedCrashReport** class utilizes **DbgHelp** utility library distributed as part of the Microsoft Debugging Tools for Windows* package.

The full set of steps required to enable **AdvancedCrashReport** is as follows:

1. Download and install the latest version of Microsoft Debugging Tools for Windows* package

2. Define **ADP_ADVANCED_CRASHREPORT** pre-processor symbol in the application project file
3. Include "**AdvancedCrashReport.h**" header file with your application
4. Add **dbghelp.lib** library to the list of libraries the application links with
5. The system **DbgHelp.dll** is available on most Microsoft Windows* installations, however on the older systems its version might not be compatible with the version of the DbgHelp.lib, which was linked to the application. To ensure that the correct version of the DbgHelp.dll is available at runtime on the end user's system developers may consider redistribution of DbgHelp.dll with their applications. Refer to the DbgHelp licensing documentation for terms and conditions of its redistribution

The following sample code demonstrates how to install AdvancedCrashReport handler to the application

```
#include "AdvancedCrashReport.h"  
...  
Application::SetCrashReport(new AdvancedCrashReport);  
...
```

On Linux* based systems, **AdvancedCrashReport** is undefined. The **DefaultCrashReport** and **CallStack** classes have been provided with the logic necessary to provide in-depth stack traces provided the application was linked with `-rdynamic`.