Version

1.0

PREEMPTIVE SOLUTIONS

PREEMPTIVE ANALYTICS

Win32 Client API User's Guide

© 2012 by PreEmptive Solutions, LLC
All rights reserved.

Version 1.0
www.preemptive.com

TRADEMARKS

PreEmptive Analytics, Runtime Intelligence, Dotfuscator, DashO, and the PreEmptive Solutions logo are trademarks of PreEmptive Solutions, LLC

Java™ is a trademark of Oracle Corporation.

.NET™ and Windows are trademarks of Microsoft, Inc.

All other trademarks are property of their respective owners.

PreEmptive Solutions, LLC has intellectual property rights relating to technology embodied in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and/or other countries.

No part of this document may be reproduced in any form by any means without prior written authorization of PreEmptive Solutions, LLC.

Contents

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Overview

Message Queuing and Transmission

Messages are not immediately sent to the server. The API queues messages and sends them when either a certain amount of time has elapsed or a number of messages have accumulated. On platforms where transmission may have performance impact, such as on mobile devices, the transmission of messages can be directly controlled by your program.

Message transmission is not guaranteed and messages may be lost. In cases of forced application shutdown or storage limitations the API may be unable to transmit queued messages or store them for off-line transmission. Messages may also be discarded if they overflow the API's queue before they can be transmitted.

Off-line Storage

Your application is not required to always have network connectivity. You can configure the API to either always store the messages locally or to store them when contact with the server is not possible. The API will automatically transmit these locally stored messages to the server when connectivity has been restored.

API Usage Pattern

The general usage pattern of the API is:

- The client creates a configuration that defines the combination of a company and an application and additional options
- The client begins its interaction with an Application Start using the configuration and an optional Flow Controller.
- The client may optionally manage Session Starts and Stops using the API or it may use the default session
- The client requests the transmission of multiple messages which include:
  - Feature Ticks to mark the occurrence of a tracked event
  - Feature Start and Stop paired messages to record the duration of a tracked event
  - System Profile to report information about the platform
  - Performance Probe to report CPU and memory usage
  - Tamper messages to indicate the client application may have been tampered
  - Expiration messages to indicate the client application is close to or has expired
  - Exception reports to indicate that the client application has encountered an exceptional condition.
- The client ends it interaction with an Application Stop

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Adding Analytics

Start up and Shut down

Until the Application Start API is called the API doesn't send any messages, and after Application Stop has been called it won't send any messages. You can always call any of the API functions, but you have to be between an Application Start and Stop to generate messages.

You will need to choose where you want to put your Application Start and Stop. While it is typical to have one of each, you can use multiple Starts and/or multiple Stops depending upon the entry point and exit points to your application. Extra calls to Start and Stop don't nest: it's the first of each one that defines starting and stopping points of your application.

Company and Application IDs

To start the API you are going to need two IDs: the Company ID and the Application ID.

If using the Runtime Intelligence endpoint, PreEmptive Solutions should have given you a company ID to use. For other endpoints, the company ID can be any valid GUID. The application ID can also be any valid GUID, but should not usually change between versions of your application.

Both of these values are GUIDs. Your Application ID represents your application across versions and platforms.

If you are using the Runtime Intelligence endpoint, it is recommended that you use a separate ID for evaluation or trial versions of your application. This is so that you can partition the data. You can combine the information from the two versions later using the Runtime Intelligence portal. This does not apply to other endpoints.

Application Start

The only thing you need to call Application Start is an instance of the Configuration. The minimal data that you need to set there is your Company and Application IDs.

C++ Example

```
#include "API.h"

Configuration* configuration = new Configuration();
configuration->SetCompanyID(L"21EC2020-CCCC-1069-A2DD-08002B30309D");
configuration->SetApplicationID(L"21EC2020-AAA1-1069-A2DD-08002B30309D");
API::ApplicationStart(configuration);
```

C Example

```
#include "riwin32.h"

PVOID configuration = CreateConfiguration();
SetCompanyID(configuration, L"21EC2020-CCCC-1069-A2DD-08002B30309D");
SetApplicationID(configuration, L"21EC2020-AAA0-1069-A2DD-08002B30309D");
ApplicationStart(configuration, 0, 0, 0);
```


© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Of course, there are many other configuration options you can set. Typical options include:

- **Additional Application Information**
  Name, Version, Type, Instance ID (serial number)
- **Endpoint Information**
  Location, SSL protocol
- **User Privacy Options**
  User Opt-In, Omit Personal Identifiable Data

After the configuration has been sent to the Application Start it is no longer used by the API and changing its values won't change the API's behavior. All that's left to do is free up the memory:

C++ Example

```
// We are done with the configuration so let's free it up
delete configuration;
```

C Example

```
// We are done with the configuration so let's free it up
DeleteConfiguration(configuration);
```

Application Stop

Just before your application terminates you should make a call to Application Stop. This generates the messages that are used to calculate the duration of the user's interaction with your application and handles any messages queued in memory.

C++ Example

```
#include "API.h"

API::ApplicationStop();
```

C Example

```
#include "riwin32.h"

ApplicationStop(0, 0);
```

As you can see from the C Example's signature there are several optional arguments you can pass to the Application Stop as well as to most of the other API functions. These are covered in the Sending Custom Data and Information About Your Application sections.

Tracking Feature Use

The most common thing do with the API is to track the use of features in your application. You can track the occurrence of a feature using Feature Ticks or you can use Feature Stop and Feature Start to track the occurrence and duration.

Adding Feature Ticks

The Feature Tick generates a message that the endpoint tallies up. All you need to provide a name that defines the feature:

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C++ Example

```
void Pascal::ConstructTriangle(int rows)
{
    API::FeatureTick(L"ConstructTriangle");
    // ...
}
```

C Example

```
void ConstructTriangle(int rows)
{
    FeatureTick(L"ConstructTriangle", 0, 0, 0);
    // ...
}
```

You can place the call to Feature Tick anywhere in the code including inside conditionals. A function can contain any number of Feature Ticks, as well as Starts and Stops, depending upon how it is used. There does not have to be a one-to-one mapping of functions to features.

Adding Feature Starts and Stops

A pair of Feature Start and Stops is used to measure not only the occurrence of a feature use but also its duration. One possible way to use it is to bracket the entire body of the function with the Start and Stop:

C++ Example

```
void Pascal::ConstructTriangle(int rows)
{
    API::FeatureStart(L"ConstructTriangle");
    // ...
    API::FeatureStop(L"ConstructTriangle");
}
```

C Example

```
void ConstructTriangle(int rows)
{
    FeatureStart(L"ConstructTriangle", 0, 0, 0);
    // ...
    FeatureStop(L"ConstructTriangle", 0, 0, 0);
}
```

It is important that you use the exact, same name in the Start and Stop calls otherwise it will look like two separate features.

The Start and Stop can also be used across multiple functions:

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C++ Example

```
void Pascal::PrintHeader()
{
    API::FeatureStart(L"PrintIt");
    // ...
}

void Pascal::PrintRows()
{
    // ...
    API::FeatureStop(L"PrintIt");
}
```

C Example

```
void PrintHeader()
{
    FeatureStart(L"PrintIt", 0, 0, 0);
    // ...
}

void PrintRows()
{
    // ...
    FeatureStop(L"PrintIt", 0, 0, 0);
}
```

Sending Custom Data

You can send custom data to the server with feature information or any other types of messages. To send over the data you construct an ExtendedKeys structure and add information to it. You can then pass this in as an argument to the message calls:

C++ Example

```
#include "API.h"

bool Pascal::ConstructTriangle(int rows)
{
    bool result = false;
    ExtendedKeys* keys = new ExtendedKeys();
    keys->Add(L"rows", n);
    API::FeatureStart(L"ConstructTriangle", keys);
    // ...
    keys->Add(L"result", result ? L"true" : L"false");
    API::FeatureStop(L"ConstructTriangle", keys);
    delete keys;
    return result;
}
```

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C Example

```
#include "riwin32.h"

int ConstructTriangle(int rows)
{
    int result = 0;
    PVOID keys = CreateExtendedKeys();
    AddExtendedKeysInt(keys, L"rows", rows);
    FeatureStart(L"ConstructTriangle", 0, 0, keys);
    // ...
    AddExtendedKeysString(keys, L"result", result ? L"true" : L"false");
    FeatureStop(L"ConstructTriangle", 0, 0, keys);
    DeleteExtendedKeys(keys);
    return result;
}
```

Values for keys can be either numeric or strings. If you are sending your data to Runtime Intelligence, then the portal will compute some basic statistics for numeric data and provide frequency counts for string values. You can use the same ExtendedKeys structure as many times as you need. If you set a key with the same name a second time its original value is replaced.

ExtendedKeys have some limits. The length of the name for the key is limited to 2000 characters and the value part for strings is limited to 4000 characters. Numeric values have up to 18 digits of precision with 5 digits to the right of the decimal point.

Once you are done with the structure you will need to dispose of it as in the previous example.

Getting System Information

The API can generate two types of system information: a system profile; performance information.

System Profile

The System Profile generates information about the operating system, its environment, and the hardware. Generating the profile is just calling the other APIs:

C++ Example

```
#include "API.h"

Configuration* configuration = new Configuration();
//…
API::ApplicationStart(configuration);
API::SystemProfile();
```

C Example

```
#include "riwin32.h"

PVOID configuration = CreateConfiguration();
//…
ApplicationStart(configuration, 0, 0, 0);
SystemProfile(0, 0, 0);
```


© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

It's recommended that you only generate one system profile per application run and that is typically done after the Application Start. The System Profile sends back information many different aspects of the machine running your application:

- **Processors** – Number of processors, clock speeds, manufacturer, and processor ID
- **Logical Disks** – Number of logical disks, volume name, size, free space, file system
- **Memory** - Speed, capacity
- **Network Adapters** - IP address, MAC address
- **Domain** - Domain name and role
- **Display** - Name, refresh rate, vertical and horizontal resolution
- **Video** - Name, memory size, color depth
- **Terminal Services** - Connections allowed
- **Sound** – Name, manufacturer
- **Modem** – Model, device type

There are two values in the Configuration that affect the data reported in the profile:

- **Full Data**
  The default is to gather as much information as possible. If Full Data is set to false the API will either skip the reporting on some time expensive data or use substitute calls that take less time but may not contains as much information.
- **Omit Personal Info**
  The default is to report on information that could identify the user and their machine. If Omit Personal Info is set to false the API will skip or mask values such as IP addresses, MAC identifiers, and names that could be used to identify the user and their machine.

Performance Probe

In contrast to the System Profile the Performance Probe is designed to be used in multiple places and called many times in your application. The Performance Probe includes a name so you can track performance information in multiple places in your application:

C++ Example

```
#include "API.h"

bool Pascal::ConstructTriangle(int rows)
{
    bool result = false;
    ExtendedKeys* keys = new ExtendedKeys();
    keys->Add(L"rows", n);
    API::FeatureStart(L"ConstructTriangle", keys);
    API::PerformanceProbe(L"ConstructTriangle.Starting", keys);
    // ...
    API::PerformanceProbe(L"ConstructTriangle.Finished", keys);
    keys->Add(L"result", result ? L"true" : L"false");
    API::FeatureStop(L"ConstructTriangle", keys);
    delete keys;
    return result;
}
```

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C Example

```
#include "riwin32.h"

int ConstructTriangle(int rows)
{
    int result = 0;
    PVOID keys = CreateExtendedKeys();
    AddExtendedKeysInt(keys, L"rows", rows);
    FeatureStart(L"ConstructTriangle", 0, 0, keys);
    PerformanceProbe(L"ConstructTriangle.Starting", 0, 0, keys);
    // ...
    PerformanceProbe(L"ConstructTriangle.Finished", 0, 0, keys);
    AddExtendedKeysString(keys, L"result", result ? L"true" : L"false");
    FeatureStop(L"ConstructTriangle", 0, 0, keys);
    DeleteExtendedKeys(keys);
    return result;
}
```

The Performance Probe only reports on two pieces of information: the CPU usage percentage; the amount of memory allocated by the program.

Reporting Exceptions

The API provides a simple way to report exceptional conditions in your application. The exception reports can be used to track exceptions reported by your application or from third party software. The report can also have user added information added to it to aid support staff. And of course you can always add Extended Key information to track application state.

Exception Event Types

Exception events come in three types:

- **Caught**
  This represents an exception that is caught by your application. Your code can either be dealing with the exception or is going to pass it up the execution chain. This is the mostly commonly used event type and is the default type of the ExceptionInfo class.
- **Thrown**
  This represents an exception that your application is creating. You would use this type for an exception event that is atypical for your application.
- **Uncaught**
  This type is usually used in catch-all handlers and indicates that your application encountered an exception it was not prepared to catch.

Again, caught is the default event type. In the following example we will show you how to change the event type.

By default, the API also sends along a list of all components loaded by the application reporting the exception. This behavior can be overridden by passing in a value of “false” for the report_appcomponent parameter of the ReportException function.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Reporting Details

To pass all the information about the exception we use an ExceptionInfo instance. For

C++ Example

```
// New one up
ExceptionInfo* pInfo = new ExceptionInfo();
// -- or --
// Create one on the stack
ExceptionInfo info;
// Use it, then free it
delete pInfo;
```

C Example

```
// Create one
PVOID pInfo = CreateExceptionInfo();
// Use it, then free it
DeleteExceptionInfo(pInfo);
```

Before telling the API to report the exception we need to fill in some details. First we need to indicate the event type as previously described – remember that the default event type is "caught". You also need to indicate the exception type and probably a message:

C++ Example

```
info.SetType(L"Illegal Value");
info.SetMessage(L"x less than zero");
```

C Example

```
SetExceptionInfoType(L"Illegal Value");
SetExceptionInfoMessage(L"x less than zero");
```

To indicate the origin of the exception, create a BinaryInfo that will accompany the ExceptionInfo. The type and method name of the BinaryInfo will be added to the stack information of the exception report.

In C++ you can also use method chaining and std::exception instances to fill in the details:

C++ Example

```
info.Caught(err).Type(L"Logic Error");
```

Adding User Details

Many GUI programs can display a message box indicating the error that has occurred to the user. This is an opportunity to have the user enter some additional information that developers can use to diagnose the problem. The ExceptionInfo provides a way to attach up to 500 characters of text to the exception report that the user can used to describe the circumstances of the event. It also has a way to pass in contact information, such as a registered user name, email address, or full mailing information back to the server.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C++ Example

```
::GetWindowText(hEmail, email, 500);
::GetWindowText(hComment, comment, 500);
info.SetUserContact(email);
info.SetUserComment(comment);
```

C Example

```
GetWindowText(hEmail, email, 500);
GetWindowText(hComment, comment, 500);
SetExceptionInfoUserContact(pInfo, email);
SetExceptionInfoUserComment(pInfo, comment);
```

Sending The Report

Now we that we have talked about all the pieces let's look at some examples of actually sending the exception report:

C++ Example

```
BinaryInfo binary;
binary.SetClassName(L"Pascal");
binary.SetMethodName(L"ConstructTriangle");
try
{
    // complicated code here
}
catch(std::exception& err)
{
    ExceptionInfo info;
    API::ReportException(info.Caught(err), &binary);
}
catch(...)
{
    ExceptionInfo info;
    API::ReportException(info.Uncaught(), &binary);
}
```

In this example we show how to report both handled and unhandled exceptions in a try/catch.

When working with C you need to set more of the information yourself:

C Example

```
GetWindowText(hControl, data, 100);
if(GetLastError() != ERROR_SUCCESS)
{
    PVOID pInfo = CreateExceptionInfo();
    SetExceptionInfoMessage(pInfo, L"GetWindowText() failed!");
    ReportException(pInfo, 0, 0, 0, TRUE);
    DeleteExceptionInfo(pInfo);
}
```


© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Advanced Configuration

User Opt-In

Your application should allow user to decide if analytics data is gathered or not. The API provides two ways to control the user opting-in to analytics gathering.

The first way to tell the API about the users opt-in is via the Configuration. You use this way when your application stores their previous selection somewhere:

C++ Example

```
#include "API.h"

Configuration* configuration = new Configuration();
configuration->SetOptIn(GetSavedOptInState());
//…
API::ApplicationStart(configuration);
```

C Example

```
#include "riwin32.h"

PVOID configuration = CreateConfiguration();
SetInitialOptIn(configuration, GetSavedOptInState());
//…
ApplicationStart(configuration, 0, 0, 0);
```

The opt-in value in the Configuration changes if the API does an application starts up right away. If it is true then the API starts right up. If it is false then the API makes a copy of the Configuration for later use, but doesn't start up, since no messages can be sent.

That brings us to the second way the opt-in can be changed and this is the runtime way by talking directly to the API. You would use this when your application would have a dialog that lets the user change their opt-in status after you have done your Application Start:

C++ Example

```
API::SetOptIn(::SendMessage(optIn, BM_GETSTATE, 0, 0) == BST_CHECKED));
```

C Example

```
SetOptIn(SendMessage(optIn, BM_GETSTATE, 0, 0) == BST_CHECKED));
```

If the opt-in state changes from false to true the API will do an automatic Application Start and analytics will start being gathered.

There are a couple of things to note about the opt-in state. If the Application Start is skipped because of the opt-in state, the API may still send messages that were previously gather and stored offline.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Once it is done sending that data it will go back to the stopped state. If you change the opt-in state from true to false the API does not immediately shut down. Instead it stops analytics gathering, by ignoring requests like Feature Tick et. al., and will send any previously created messages. It will also send the Application Stop message once that is called.

Information About Your Application

There is much more information you can send to the configured endpoint about your application than just its application ID.

- **Name**
  The name of the application. The portal uses this as the default display name for the application, but you can rename it for display purposes.
- **Version**
  A version string for the application. Although no specific format is defined a dotted numeric representation is recommended.
- **Type**
  A user defined application type.

These values along with the Application ID are used to define a particular application in the Runtime Intelligence Portal. The only value that you should change over time is the Version string.

C++ Example

```
Configuration* configuration = new Configuration();
configuration->SetCompanyID(L"21EC2020-CCCC-1069-A2DD-08002B30309D");
configuration->SetApplicationID(L"21EC2020-AAA1-1069-A2DD-08002B30309D");
API::ApplicationStart(configuration);
```

C Example

```
PVOID configuration = CreateConfiguration();
SetCompanyID(configuration, L"21EC2020-CCCC-1069-A2DD-08002B30309D");
SetApplicationID(configuration, L"21EC2020-AAA0-1069-A2DD-08002B30309D");
ApplicationStart(configuration, 0, 0, 0);
```

There is one more application value that you can use to identify a particular copy of your application.

- **Instance ID**
  An identifier for the instance of the application, such as a serial number. No specific format is defined.

C++ Example

```
configuration->SetApplicationInstanceID(GetSerialNumber());
```

C Example

```
SetApplicationInstanceID(GetSerialNumber());
```


© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Storing Data Offline

By default the API will try to send messages directly to the configured endpoint. If there is no internet connection or there is a problem reaching the server it will save the data locally for later transmission.

You can control this behavior with the Configuration instance passed to Application Start. The settings that control the APIs behavior are:

- **Offline**
  This indicates that the API will not attempt to send messages to the server. By default the API is "on-line".
- **SupportOfflineStorage**
  This tell the API if it is allowed to store data locally. By default the API will try to store data locally.

Both of these are permanent conditions for the duration of the application run. If you set both of these values to false your call to Application Start will return false indicating the API did not start up.

There is also a control called **SendDisabled**. An initial value for it can be set on the Configuration and then you can change its setting while the application run is active. This differs from Offline in that it is a temporary condition. We will talk more about that in Controlling the Transmission Window.

Controlling Data Transmission

The API doesn't send messages immediately to the server, instead it bundles them up into groups. The number of message that make up a group is based on a combination of the maximum size of the message queue, how quickly your application is generating messages, and how long you want to keep messages in memory. All of this behavior is determined by the Flow Controller passed into Application Start.

The Flow Controller

The first items we will look at in the Flow Controller are those controlling the in-memory queue size. This is where messages are held before being sent to the server or to local storage. There are two controls:

- **QueueSize**
  This is the absolute maximum number of messages that will be held in memory. If the queue overflows the oldest messages will be discarded.
- **HighWater**
  This is the point in the queue where the API will try to bundle up the messages for transmission or storage. The space between the HighWater and the QueueSize gives the API room to handle applications sending messages at a rapid pace.

When you set the QueueSize the HighWater is automatically adjusted to be 1/3 less than the size. You may want to adjust these values if your application can send rapid bursts of messages.

The next two items control how often the API will try to flush the queue regardless of how many messages have accumulated.

- **MaximumInterval**
  How long, in milliseconds, will a message sit in the queue before the API will try to transmit it or store it locally.
- **MinimumInterval**
  The minimum time, in milliseconds, that the API will wait between queue checks.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Adjusting these values requires care. Having a MaximumInterval that is too large can cause the API to hang on to memory used to store messages – this can be exacerbated by having a large QueueSize. Setting it too small can steal time from your application.

The actual time between checks is determined by the FlowController and the rate at which your application is sending messages. This brings us to the next two controls:

- **Gain**
  The rate at which the queue checking interval is adjusted when there are messages in the queue. Setting the Gain to 100 means 100% - adjust the interval based on the determined message rate. Using values less than 100 slowly adjust the value which values greater than 100 lets you anticipate increasing message rates. The default value is 66.
- **QuietGain**
  The rate at which the queue checking interval is adjusted when there are no messages in the queue. This value controls how quickly the interval works its way back to MaximumInterval when there are no queued messages. You can use values from 1 to 100. The default value is 33.

Finally, there are two controls that let the FlowController handles transmission problems:

- **MaximumSequentialFailureCount**
  The FlowController keeps track of the number of times the server cannot be contacted due to networking issues. Every time the Server is contacted this count is reset. When the count exceeds the set value the network is considered down for a period of time. This keeps the API from trying to use network resources when the chance of success is low. The default value is 3.
- **RetryTimeout**
  Once we reach the failure count the API determines a retry time based on this value. It won't try to send to the server again until this timeout has elapsed. The default value is 60 seconds.

Controlling the Transmission Window

The Flow Controller uses the previously mentioned values to adjust how often it will empty the in memory queue of messages. Although this takes place on a background thread with a lower priority it still can happen at a time where your application is doing something critical. The API provides a way to control when the transmission can take place.

If you have places in your application where you need to prevent the API from sending messages you can use the Send Disabled feature:

C++ Example

```
API::SetSendDisabled(true);
// Time critical code here…
API::SetSendDisabled(false);
```

C Example

```
SetSendDisabled(1);
// Time critical code here…
SetSendDisabled(0);
```

Another way to use Send Disabled is only allow sending when you application is in an idle state, for example sitting at a menu screen. In this case you would want to start up the API in the disabled state, which you can do with the Configuration instance:

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

C++ Example

```
Configuration* configuration = new Configuration();
configuration->SetSendDisabled(true);
API::ApplicationStart(configuration);
// When you enter an idle state
API::SetSendDisabled(false);
// When you are busy again
API::SetSendDisabled(true);
```

C Example

```
PVOID configuration = CreateConfiguration();
SetInitialSendDisabled(configuration, 1);
ApplicationStart(configuration, 0, 0, 0);
// When you enter an idle state
SetSendDisabled(0);
// When you are busy gain
SetSendDisabled(1);
```

The last way that you can control the sending of messages is by telling the API that it should flush any queued messages right away. This is done with the **SendMessages** method. You can call this method if you have sending enabled or disabled: any messages that are in memory are sent.

You can keep your application in the Send Disabled state permanently and flush the messages when your application becomes idle:

C++ Example

```
Configuration* configuration = new Configuration();
configuration->SetSendDisabled(true);
configuration->SetQueueSize(300);
API::ApplicationStart(configuration);
// When you enter an idle state
API::SendMessages();
```

C Example

```
PVOID configuration = CreateConfiguration();
SetInitialSendDisabled(configuration, 1);
SetFlowControllerQueueSize(configuration, 300);
ApplicationStart(configuration, 0, 0, 0);
// When you enter an idle state
SendMessaages();
```

If you decide to use this technique then messages are held in memory until you can **SendMessages**. Because of this the **QueueSize** is set rather large so that we will not have an overflow and drop messages.

Managing Multiple Sessions

Each Application Start begins a default session which is ended by its matching Application Stop. If your application has the concept of the user's session being the entire run of the application you can ignore sessions and just pass in NULL for the session ID. The API will record all the events on the default session automatically.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

If you are adding analytics to an application that has multiple users, such as a server, you will probably want to have multiple session IDs.

Session IDs are GUIDs in string form. All of the API calls, except for Application Start and Stop take a session ID as an argument.

C++ Example

```
// When a new user is connected create and store their ID
ConnectUser();
API::SessionStart(GetUserSessionID());
// Record some features using the current ID
API::FeatureTick(L"ConstructTriangle", GetUserSessionID());
// When a user logs out
API::SessionStop(GetUserSessionID());
// And clear it form storage
DisconnectUser();
```

C Example

```
// When a new user is connected create and store their ID
ConnectUser();
SessionStart(GetUserSessionID());
// Record some features using the current ID
FeatureTick(L"ConstructTriangle", GetUserSessionID());
// When a user logs out
SessionStop(GetUserSessionID());
// And clear it form storage
DisconnectUser();
```

Your implementation of ConnectUser and DisconnectUser would manage the binding of the user to the session ID which GetUserSessionID returns it for use by the API functions.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

API Language Interoperability

Analytics can be added to applications created in multiple languages. The PreEmptive Analytics API is available for several languages, but only one can control the Application Start and Stop at a time. The API that starts first controls the scope of the application run and has APIs implemented in other language run in a subordinate fashion. This is done by passing in some internal identifiers from the controlling API to the subordinate one:

C++ Example

```
// If we are the controlling API-
API::ApplicationStart(configuration);
StartOtherAPI(API::GetMessageGroup(), API::GetDefaultSession());

// If we are the subordinate API-
configuration->SetDefaultIDs(otherMessageGroup, otherDefaultSession);
API::ApplicationStart(configuration);
```

C Example

```
// If we are the controlling API-
ApplicationStart(configuration);
StartOtherAPI(GetMessageGroup(), GetDefaultSession());

// If we are the subordinate API-
SetDefaultIDs(configuration, otherMessageGroup, otherDefaultSession);
ApplicationStart(configuration, 0, 0, 0);
```

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Advanced Messages

Tamper Messages

PreEmptive Analytics can be used to track suspected tampering of your application. If your tamper checking code detects that the application has been altered you can report that to the server:

C++ Example

```
if(!ChecksumMatches())
{
    API::TamperEvent();
}
```

C Example

```
if(!ChecksumMatches())
{
    TamperEvent(0, 0, 0);
}
```

Tamper detection code can be automatically added to your .NET application by Dotfuscator or to your Java application by DashO.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED

Expiration Messages

PreEmptive Analytics can be used to track life cycle events on applications that are time locked. You can report on subscription or evaluation copies that are nearing or have reached an expiration date. Both require a Shelf Life ID that you can get from PreEmptive Solutions.

C++ Example

```
TCHAR* shelfLifeID = L"a83fd44b-d5d2-4171-92ce-d878c8f82113";
int daysLeft = GetEvalDaysLeft();
if(daysLeft <= 0)
{
    API::ExpiredEvent(shelfLifeID);
    // Display a dialog, exit the application
}
else
{
    if(daysLeft <= 7)
    {
        API::ExpirationWarningEvent(shelfLifeID);
        // Display a dialog
    }
}
```

C Example

```
TCHAR* shelfLifeID = L"a83fd44b-d5d2-4171-92ce-d878c8f82113";
int daysLeft = GetEvalDaysLeft();
if(daysLeft <= 0)
{
    ExpiredEvent(shelfLifeID, 0, 0, 0);
    // Display a dialog, exit the application
}
else
{
    if(daysLeft <= 7)
    {
        ExpirationWarningEvent(shelfLifeID, 0, 0, 0);
        // Display a dialog
    }
}
```

Expiration code can be automatically added to your .NET application by Dotfuscator or to your Java application by DashO.

© 2012 PREEMPTIVE SOLUTIONS LLC. ALL RIGHTS RESERVED