



## **Platform SDK 8.1**

# Developer's Guide Wiki Redirect

ALERT: This document is available as a PDF only to support searches from the Technical Support Knowledge Base. Click here ([Platform SDK Developer's Guide](#)) to be redirected to the content in its original format.

# Contents

## Articles

Platform SDK Developer's Guide	1
Developers Guide PDF - PSDK 7.6	3
LCA Hang-Up Detection Support	3
Lazy Parsing of Message Attributes	8
Using the Switch Policy Library	10

## References

Article Sources and Contributors	20
Image Sources, Licenses and Contributors	21

# Platform SDK Developer's Guide

---



Home > Platform SDK > Platform SDK Developer's Guide



**Purpose:** To provide details, recommendations, and code samples that will aid developers using the Platform SDK.



**Important Note:** The Platform SDK Developer's Guide information in this wiki is currently a work-in-progress. Check back in the following days for new articles and additional content pages to be added.

## Description

This document introduces you to the tools and examples provided to help you get started with Platform SDK development.

The Platform SDK is divided broadly into three categories:

- Application Blocks provide production-ready blocks of code you should leverage, and modify if necessary, when creating applications.
- Server-Specific SDK Protocols work directly with Genesys servers using message-based requests and events.
- Library Components offer additional features and functionality such as logging.

There is also a list of additional topics that may pertain to general developer topics or common SDK functionality.

For additional information about the Platform SDKs, please check the introductory materials provided as part of the Platform SDK API Reference for your release.

## Application Blocks

Application Block	Releated Documentation
Application Template Application Block	
Configuration Object Model Application Block	
Message Broker Application Block	
Protocol Manager Application Block	
Warm Standby Application Block	

## Server-Specific SDK Protocols

Platform SDK Protocol	Genesys Server(s)	Releated Documentation
Configuration Platform SDK	Configuration Server	
Contacts Platform SDK	Universal Contact Server	
Management Platform SDK	Message Server Solution Control Server Local Control Agent	LCA Hang-Up Detection Support
Open Media Platform SDK	Interaction Server	
Outbound Contact Platform SDK	Outbound Contact Server	
Routing Platform SDK	Universal Routing Server Custom Server	
Statistics Platform SDK	Stat Server	
Voice Platform SDK	T-Servers	
Web Media Platform SDK	Chat Server E-Mail Server Java Callback Server	

## Library Components

Library Component	Releated Documentation
Platform SDK Log Library	
Platform SDK Switch Policy Library	Using the Switch Policy Library

## Additional Topics

- Lazy Parsing of Message Attributes

## New Content by Release

This section provides a quick outline of developer content based on the release where that information first became relevant, or where it where last updated.

### Release 8.1.x

New Features:

- Lazy Parsing of Message Attributes
- LCA Hang-Up Detection Support

### Release 8.0

- Please refer to developer information provided as part of the introductory material in the Platform SDK API Reference for this release.

### Release 7.6

- Please refer to the Platform SDK 7.6 Developer's Guide PDF.

# Developers Guide PDF - PSDK 7.6

---



Home > Platform SDK > Platform SDK Developer's Guide > Developers Guide PDF - PSDK 7.6



76sdk\_dev\_platform.pdf <sup>[1]</sup>

## Description

This document is currently only applicable to the 7.6 release of Platform SDK.

This document introduces you to the tools and examples provided to help you get started with Platform SDK development. In brief, you will find the following information in this guide:

- Descriptions of the Application Blocks included with Platform SDK 7.6.
- Setup instructions and analysis of the code examples included with Platform SDK 7.6.

---

<pdf>[http://devzone.genesyslab.com/devportal/76%20SDK%20Documentation/76sdk\\_dev\\_platform.pdf](http://devzone.genesyslab.com/devportal/76%20SDK%20Documentation/76sdk_dev_platform.pdf)</pdf>

## References

[1] [http://devzone.genesyslab.com/devportal/76%20SDK%20Documentation/76sdk\\_dev\\_platform.pdf](http://devzone.genesyslab.com/devportal/76%20SDK%20Documentation/76sdk_dev_platform.pdf)

# LCA Hang-Up Detection Support

---



Home > Platform SDK > Platform SDK Developer's Guide > LCA Hang-Up Detection Support



**Description:** This page provides:


- an overview and list of requirements for the LCA Hang-Up Detection Support feature
- design details explaining how this feature works
- code examples showing how to implement LCA Hang-Up Detection Support in your applications

## Introduction to LCA Hang-up Detection Support

Beginning with release 8.1, the Platform SDKs now allow user-developed application to include hang-up detection functionality.

The Genesys Management Layer relies on Local Control Agent (LCA) to monitor and control applications. An open connection between LCA and Genesys applications is typically used to determine which applications are running or stopped. However, if an application that has stopped responding still has a connection to LCA then it could appear to be running correctly - preventing Management Layer from switching over to a backup application or taking other actions to restore functionality.

Hang-up detection allows Local Control Agent (LCA) to detect unresponsive Genesys applications by checking for regular heartbeat messages. When an unresponsive application is found, pre-configured actions can taken - including triggering alarms or restarting the application.

 **Note:** Hang-up detection functionality has been available in the Genesys Management Layer since release 8.0.1. For more information, refer to the Framework 8.0 Management Layer User's Guide <sup>[1]</sup>. For details about related configuration options, refer to the Framework 8.0 Configuration Options Reference Manual <sup>[2]</sup>.

Two levels of hang-up detection are available: implicit and explicit.

---

## Implicit Hang-up Detection

The easiest form of hang-up detection to implement is implicit hang-up detection.

In this scenario, application status is monitored through the connection between your application and LCA. This functionality can be extended by adding a requirement that your application periodically interacts with LCA (either responding to ping request or sending its own heart-beat messages) as a necessary condition of application liveliness.

This simple form of hang-up detection can be implemented internally by using the `LocalControlAgentProtocol` to connect to LCA. In this case, existing applications only need to be rebuilt with a version of `LocalControlAgentProtocol` that supports hang-up detection functionality - no coding changes are required - and given the appropriate configuration options in Genesys Management Layer.

## Explicit Hang-up Detection

Explicit hang-up detection offers more robust protection from applications that may become unresponsive, but is also more complex.

The periodic interaction that is monitored by implicit hang-up detection only confirms that your application can interact with LCA. In most cases this means that the application is able to communicate with other apps and that the thread responsible for communicating with LCA is still active. However, multi-threaded applications may contain other threads that are blocked or have stopped responding without interrupting communication with LCA. Explicit hang-up detection allows you to determine when only part of your application hangs-up by monitoring individual threads in the application.

In addition to allowing your application to register (or unregister) individual threads to be monitored, explicit hang-up detection also allows your application to stop or delay the monitoring process. Threads that execute synchronous functions (which can block thread execution for some extended periods) or other features that prevent accurate monitoring should take advantage of this feature.

## Feature Overview

- To maintain backwards compatibility, hang-up detection must be explicitly enabled in the application configuration.
- Implicit hang-up detection can be used for applications that do not require complex monitoring functionality. No code changes are required, just rebuild your application using the new version of `LocalControlAgentProtocol`.
- Explicit hang-up detection requires minimal application participation - enabling monitoring, registering and unregistering execution threads, and providing heartbeats. Most hang-up detection functionality is implemented within the Management Layer component, while all timing information (such as maximum allowed period between heartbeats) is configured through Genesys Management Layer.

## System Requirements

Genesys Management Layer:

- Release 8.0.1 or later

Platform SDK for .NET:

- Management SDK protocol release 8.1 or later
- .NET Framework 3.5
- Visual Studio 2008 (required for .NET project files)

Platform SDK for Java:

- Management SDK protocol release 8.1 or later
-

- J2SE 5.0 or Java 6 SE runtime

## Design Details

This section provides an overview of the main classes and interfaces used to add thread monitoring functionality for Explicit hang-up detection. Before using the classes and methods described here, be sure that you have implemented basic LCA Integration in your application using `LocalControlAgentProtocol`.

Although the details of thread monitoring implementation are slightly differently for Java and .NET, the basic idea is the same: to create and update a thread monitoring table that LCA can use to confirm the status of your application.

Note that for implicit hang-up detection you are only required to rebuild your application and make adjustments to the configuration options in Genesys Management Layer; the details described below are not required for simple application monitoring.

## Thread Monitoring Table


The new thread monitoring functions described below allow `LocalControlAgentProtocol` to create and maintain a thread monitoring table within the application. This table tracks basic thread status.

**Sample Thread Monitoring Table**

OS Thread ID	Logical Thread ID	Thread Class	Heartbeat Counter	Flags
0	«main»	1	444345	active
1	«pool_1»	2	354354	suspend
2	«pool_2»	2	432432	deleted
3	«pool_3»	2	434323	active
4	«DB_store»	3	31212	active
....	....	....	....	....

Each row corresponds to a monitored thread. Columns of the table are:

- OS Thread ID—The OS-specific thread ID, used for thread identification during monitoring. OS thread ID is not passed by application but is received directly from system.
- Logical Thread ID – Application logical thread ID (or logical name, in Java). Used for logging and thread identification.
- Thread Class—Thread class integer. This value is only meaningful within the scope of the application; threads with the same thread class value in a different application can have different roles. Examples of thread classes might be the main loop thread, pool threads, or special threads (such as external authentication threads in `ConfigServer`).
- Heartbeat Counter—Cumulative counter of `Heartbeat()` calls made by the corresponding thread. Incrementing this value is the main way to indicate that the thread is still alive.

 **NOTE:** This value is initialized with a random value when the thread is registered for monitoring. This prevents incorrect hang-up detection if threads are created and terminated with high frequency, leading to repeating OS thread IDs.

- Flag—Special flags.
  - Suspended/Resumed—Corresponds to the state of thread monitoring.
  - Deleted—Used internally to notify LCA that a thread was unregistered from monitoring.

## .NET Implementation

### ThreadMonitoring Class

The `ThreadMonitoring` class is defined in the `Genesyslab.Diagnostics` namespace of `Genesyslab.Core.dll`. This class contains the following public static methods:

- `Register(int threadClass, string threadLogicId)`—enables monitoring for this thread
- `Unregister()`—removes this thread from monitoring
- `Heartbeat()`—increases heartbeat counter for this thread (indicating that thread is still alive)
- `SuspendMonitoring()`—suspend monitoring for this thread
- `ResumeMonitoring()`—resumes monitoring for this thread

 **Note:** Each method should be called from within the thread that is being monitored.

When a thread is registered for monitoring, the following parameters are included:

- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.
- `threadLogicId`—A logical, descriptive thread ID that is independent from thread ID provided by OS. This value is used for thread identification within LCA and for logging purposes. This ID should be unique within the application.


### PerformanceCounter Constants

The following String constants (names) are defined in the `ThreadMonitoring` class:

```
public const string CategoryName = "Genesyslab PSDK .NET";
public const string HeartbeatCounterName = "Thread Heartbeat";
public const string StateCounterName = "Thread State";
public const string ProcessIdCounterName = "ProcessId";
public const string OsThreadIdCounterName = "OsThreadId";
```

The Platform SDK thread monitoring functionality uses these constants to manage `PerformanceCounter` values. In addition to these custom performance counters, you can also use standard ones, such as those defined in `Thread` category: "% Processor Time", "% User Time", etc.

See MSDN<sup>[3]</sup> for details about performance counters.

 **Note:** Use of `PerformanceCounters` is optional, and is not required for LCA hang-up detection functionality.

## Java Implementation

### ThreadHeartbeatCounter class

The `ThreadHeartbeatCounter` class is defined in the `com.genesyslab.platform.commons.threading` package, located within `commons.jar`. This class is designed as a JMX<sup>[4]</sup> MBean and implements the public `ThreadHeartbeatCounterMBean` interface which is accessible through Java management framework.

There is no public constructor for the `ThreadHeartbeatCounter` class; each thread that you want to monitor should create its own instance with following static method:

```
public static ThreadHeartbeatCounter createThreadHeartbeatCounter(
    String threadLogicalName,
    int threadClass);
```

When a thread is registered for monitoring, the following parameters are included:



- `threadLogicalName`—A logical, descriptive thread name that is used to identify the thread within LCA and for logging purposes. This name should be unique within the application.
- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.

One key difference from thread monitoring using .NET is the need to create a monitoring object instance. The lifecycle of this object, including `MBeanServer` registration, is supported by the parent class `PSDKMBeanBase` and is shown in the five steps below:

1. Start monitoring a thread:

```
ThreadHeartbeatCounter monitor =  
    ThreadHeartbeatCounter.createThreadHeartbeatCounter(  
        threadId, threadClass);  
monitor.initialize();
```

1. Notify LCA that thread is still alive (increase heartbeat counter):

```
monitor.alive();
```

1. Suspend monitoring of this thread:


```
monitor.setActive(false);
```

1. Resume monitoring of this thread:

```
monitor.setActive(true);
```

1. Finish monitoring and unregister this thread:

```
monitor.unregister();
```

 **Note:** Each of these methods must be called from within the thread that is being monitored.

Once a `ThreadHeartbeatCounter` object is unregistered, that instance cannot be reused. To begin monitoring that thread again (or any other) you first need to create a new instance of the a thread monitoring object.

### **ThreadHeartbeatCounterMBean interface**

The `ThreadHeartbeatCounterMBean` interface is intended to present an open API to the JMX MBean. This interface contains the following publicly accessible methods:

```
public long getThreadSystemId();  
public String getLogicalName();  
public int getThreadClass();  
public void setThreadClass(int newThreadClass);  
public int getHeartbeatCounter();  
public void setActive(boolean isActive);  
public boolean isActive();
```

These methods are "MBean client-side" methods and are used by LCA protocol to get actual information about the thread for the monitoring table. They also allow users to change the thread class and suspend or resume thread monitoring (using `setActive(false/true)`) of a particular thread at application runtime.

## References

- [1] <http://genesyslab.com/support/dl/retrieve/default.asp?item=B8C93DA63FA831AA33AC3542BCCE384C&view=item>
- [2] <http://genesyslab.com/support/dl/retrieve/default.asp?item=B5C334AA30A63CA2389E046E95E2145F&view=item>
- [3] MSDN PerformanceCounter Class (<http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx>)
- [4] JMX: Java Management Extensions (<http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/>)

# Lazy Parsing of Message Attributes

---



Home > Platform SDK > Platform SDK Developer's Guide > Lazy Parsing of Message Attributes



**Description:** This page provides:

- an overview and list of requirements for the lazy parsing feature
- design details explaining how this feature works
- code examples showing how to implement lazy parsing in your applications

## Introduction to Lazy Parsing

Lazy parsing allows users to specify which attributes should always be parsed immediately, and which attributes should be parsed only on demand.

Some complex attributes (such as the ConfObject attribute found in some ConfigServer protocol messages) are large and very complex. Unpacking these attributes can be time-consuming and, in cases when an application is not interested in that data, can affect program performance. This issue is resolved by using the "lazy parsing" feature included with the Platform SDK 8.1 release, which is described in this article.

When this feature is turned off, all message attributes are parsed immediately - which is normal behavior for previous version of the Platform SDK. When lazy parsing is enabled, any attributes that were tagged for lazy parsing are only parsed on demand. In this case, if the application does not explicitly check the value of an attribute tagged for lazy parsing then that attribute is never parsed at all.

## Feature Overview

- Platform SDK includes configuration options to turn the lazy parsing functionality on or off for each individual protocol that supports this feature.
  - Potentially time-consuming attributes that are candidates for lazy parsing are selected and marked by Platform SDK developers. Refer to your Platform SDK API Reference for details.
  - To maintain backwards compatibility, there is no change in how user applications access attribute values.
  - By default, the lazy parsing feature is turned off.
-

## System Requirements

Platform SDK for .NET:

- Management SDK protocol release 8.1 or later
- .NET Framework 3.5
- Visual Studio 2008 (required for .NET project files)

Platform SDK for Java:

- Management SDK protocol release 8.1 or later
- J2SE 5.0 or Java 6 SE runtime

## Design Details

This section describes the main classes and interfaces you will need to be familiar with to implement lazy parsing in your own application. For illustration purposes, .NET code snippets are provided.

### Enabling and Disabling the Lazy Parsing Feature

At any time, a running application can enable or disable lazy parsing for a specific protocol in just a few lines of code. This is done in three easy steps:

1. Create a new `KeyValueCollection` object.
2. Set the appropriate value for the `CommonConnection.LazyParsingEnabledKey` field. A value of `True` enables the feature, while `False` disables lazy parsing.
3. Use a `KeyValueConfiguration` object to apply that setting to the desired protocol(s).

**Note:** The default value of the `CommonConnection.LazyParsingEnabledKey` field is always `False`, with the lazy parsing feature disabled.

Once lazy parsing mode is enabled for a protocol, the change is applied immediately. Every new message that is received takes the lazy parsing setting into account: parsing entire messages if the feature is disabled, or leaving some attributes unparsed until their values are requested if the feature is enabled.

To enable lazy parsing for the Configuration Server protocol, an application would use the following code:

```
KeyValueCollection kvc = new KeyValueCollection();
kvc[CommonConnection.LazyParsingEnabledKey] = "true";
KeyValueConfiguration kvcfg = new KeyValueConfiguration(kvc);
ConfServerProtocol cfgChannel = new ConfServerProtocol(endpoint);
cfgChannel.Configure(kvcfg); //lazy parsing is immediately active after this line
```

To disable lazy parsing for the protocol, only the second line of code is changed (as shown below):

```
kvc[CommonConnection.LazyParsingEnabledKey] = "false";
```

## Accessing Attribute Values

There is no difference in how applications access attribute values from returned messages. Whether the lazy parsing feature is enabled or disabled, whether the attribute being access supports lazy parsing or not, your code remains exactly the same.

However, you should consider differences in timing when accessing attribute values.

- When lazy parsing is disabled, the entire message is parsed immediately when it is received. Accessing attribute values is very fast, as the requested information is already prepared.
- When lazy parsing is enabled, the delay to parse the message upon arrival is smaller but accessing any attributes that support lazy parsing causes a slightly delay as that information must first be parsed. Note that accessing the same attribute a second time will not result in the attribute information being parsed a second time; Platform SDK saves parsed data.

## Additional Notes

- XML Serialization - The `XmlMessageSerializer` class has been updated to support lazy parsing. If a message that contains unparsed attributes is serialized, then `XmlMessageSerializer` will trigger parsing before the serialization process begins.
- ToString function - Use of the `ToString` method does not trigger parsing of attributes that support lazy parsing. In this case, each unparsed attribute has its name printed along with a value of: "<value is not yet parsed>".

# Using the Switch Policy Library

---



Home > Platform SDK > Platform SDK Developer's Guide > Using the Switch Policy Library



**Description:** This document shows how to add simple T-Server functionality to your applications by using the Switch Policy Library.

The Platform SDK Switch Policy Library (SPL) can be used in applications that need to perform agent-related switch activity with a variety of T-Servers, without knowing beforehand what kinds of T-Servers will be used. It simplifies these applications by indicating which switch functions are available at any given time and also by showing how you can use certain switch features in your applications. However, if your application works with only one kind of T-Server, you may want to have your application communicate directly with the T-Server, rather than using SPL.

## Switch Policy Library Overview

Some telephony applications need to work with more than one type of switch. Unfortunately, however, one switch may not perform a particular telephony function in the same way as another switch. This means that it can be useful to have an abstraction layer of some kind when working with multiple switches, so that you do not need custom code for each switch that is used by the application. The Switch Policy Library is designed with just this kind of abstraction in mind.

## Setting Up Switch Policy Library

SPL should be used by your agent desktop applications as a library, which means that it would be located within the agent desktop application shown above. The application can call SPL for guidance on how to send requests to or process events from your T-Server, as shown in the #Code Samples section.

---

SPL is driven by an XML-based configuration file that supports many commonly-used switches in performing agent-related functions. Your application can query SPL to determine whether a particular feature is supported for the switch you want to work with. If a feature you need is not supported for the switches you need to work with, you can make a copy of the default configuration file and modify it as needed.

**Note:** Genesys does not support modifications to the SPL configuration file. Any modifications you make are performed at your own risk.

A copy of the default configuration file is included inside the Switch Policy Library DLL. There is also a copy in the Bin directory of the Platform SDK installation package. If you need to modify the configuration file, you can use the app.config file for SPL to point to your copy.

## Code Samples

This section contains examples of how to perform useful functions with SPL.

The functions discussed here are all contained in a compilable and runnable sample application that is available on the Downloads page of the Genesys Documentation Wiki. This site also hosts the SPL IsPossible Feature Demo application. This sample application lets you specify a switch and certain characteristics of the main and secondary parties to a call, as well DN state information. Once you have done this, it will show you which functions are available for that switch, based on the characteristics you have specified. This application can be very helpful in understanding the kinds of things that are available to your application when you use SPL.

***Editor Note: Need to locate and upload these samples before publishing!***

These samples each require a valid instance of the ISwitchPolicyService, which can be created as shown here:

```
ISwitchPolicyService policyService =  
    SwitchPolicyFactory.CreateSwitchPolicyService();
```

**Note:** The DN classes specified below implement the IDNContext interface, while the Party classes implement the IPartyContext interface, and the Call classes implement the ICallContext interface.

## Get A Phone Set Configuration

On some switches, phone sets are presented as more than one Directory Number (DN). These DNs may also have different types, such as Position and Extension. Because these configurations vary by switch type, an application needs to know how the phone set configuration for a particular switch is structured. For example, it needs to know how many DNs are used to represent a phone set, and what their types are. To retrieve this phone set configuration information, perform the following steps:

1. Create an instance of PhoneSetConfigurationContext, specifying the switch type.
2. Call ISwitchPolicyService.GetPolicy, using this PhoneSetConfigurationContext.
3. Analyze the returned PhoneSetConfigurationPolicy. The PhoneSetConfigurationPolicy.Configurations property will contain all possible phone set configurations for the specified switch.

The following code snippet shows how to do this:

```
PhoneSetConfigurationContext context =  
    new PhoneSetConfigurationContext("SomeSwitch");  
PhoneSetConfigurationPolicy policy =  
    switchPolicyService.GetPolicy<PhoneSetConfigurationPolicy>(context);  
foreach (PhoneSetConfiguration configuration in policy.Configurations)  
{  
    Console.WriteLine(configuration);  
}
```

## Get Phone Set Availability Information

When working with a phone set, additional information about the included DNs may be required. This could include information about which of the DNs should be available to the end user (for example, which ones should be visible in the user interface), which of them is callable, and which number (the Callable Number) the application should use to reach the agent who is logged into the phone set. To retrieve this phone set availability information, perform the following steps:

1. Create an instance of `DNAAvailabilityContext` and populate it with the following required information:
  - Specify the switch type
  - Specify the Agent ID
  - Fill the DN collection with valid implementations of `IDNContext`
2. Call `ISwitchPolicyService.GetPolicy`, using this `DNAAvailabilityContext`.
3. Analyze the returned `DNAAvailabilityPolicy`. The `DNAAvailabilityPolicy.DNStatuses` property will contain availability information for each DN in the request.

The following code snippet shows how to do this:

```
private static void DemonstratedDNAAvailability(ISwitchPolicyService service)
{
    DNAAvailabilityContext dnacontext =
        new DNAAvailabilityContext("SomeSwitch");
    dnacontext.AgentId = "AgentLogin1000";
    dnacontext.DNs.Add(new Dn
    {
        AgentStatus = AgentStatus.Ready,
        Identifier = "1000",
        ServiceStatus = ServiceStatus.InService,
        Type = AddressType.DN
    });
    dnacontext.DNs.Add(new Dn
    {
        AgentStatus = AgentStatus.Ready,
        Identifier = "2000",
        ServiceStatus = ServiceStatus.InService,
        Type = AddressType.Position
    });

    DNAAvailabilityPolicy dnpolicy =
        service.GetPolicy<DNAAvailabilityPolicy>(dnacontext);
    DisplayInColor(dnpolicy, ConsoleColor.Red);
}
```

## Get Function Availability Information for the Current Context

Some switches differ in when they allow certain functions to be performed. Also, some functions can always be performed on certain switches, while others may be impossible to perform. For example, `RequestMergeCalls` can never be performed on some switches. For other functions, whether or not the function can be performed varies depending on context. For example, on some switches `RequestReleaseCall` can only be used when a call is in a Held, Dialing, or Established state, while on other switches it is also possible to release a call when it is in a Ringing state. In addition to this, on some switches the phone set is presented as more than one Directory Number (DN) and each

DN can have a different type, such as Position and Extension. Some functions are allowed for both types, while some other functions may be restricted to a certain DN type. To retrieve this kind of function availability information for the current context, perform the following steps:

1. Create an instance of `FunctionHandlingContext` and populate it with the following required information:
  - Specify the switch type
  - Specify the request by setting the `Message` property
  - Describe the context as fully as possible
2. Call `ISwitchPolicyService.GetPolicy`, using this `FunctionHandlingContext`.
3. Analyze the returned `FunctionAvailabilityPolicy`. If the specified request is possible in the given context, the `IsFunctionAvailable` property will be true. However, if the request is not supported, SPL will return null.

The following code snippet shows how to do this:

```
foreach (string switchType in new[] { swTypeA4400Classic, swTypeA4400emul, swTypeA4400Subs })
{
    DNContext dn = new DNContext //implements IDNContext
    {
        Identifier = "1001",
        Type = AddressType.DN,
        AgentStatus = AgentStatus.Ready,
        ServiceStatus = ServiceStatus.InService,
        DndStatus = FunctionStatus.Off,
        ForwardStatus = FunctionStatus.Off
    };

    DNContext otherDN = new DNContext
    {
        Identifier = "2001",
        Type = AddressType.DN,
        AgentStatus = AgentStatus.Ready,
        ServiceStatus = ServiceStatus.InService,
        DndStatus = FunctionStatus.Off,
        ForwardStatus = FunctionStatus.Off
    };

    foreach (CallType callType in Enum.GetValues(typeof(CallType)))
    {
        PartyContext mainParty = new PartyContext //implements IPartyContext
        {
            Identifier = "1002",
            Status = PartyStatus.Established,
            CallType = callType,
            IsConferencing = true,
            IsTransferring = true,
            DN = dn
        };

        PartyContext otherParty = new PartyContext
```

```

    {
        Identifier = "1002",
        CallType = callType,
        DN = otherDN,
        IsConferencing = true,
        IsTransferring = true,
        Status = PartyStatus.Established
    };

    CallContextStub ccontext = new CallContextStub //implements ICallContext
    {
        CallType = callType,
        Destination = mainParty,
        Origination = otherParty,
        Identifier = "1002",
        IsConferencing = true,
        IsTransferring = true,
        Parties = new List<IPartyContext>{mainParty,otherParty},
        Parent = null//no parentCall - our call is solitary call.
    };

    FunctionHandlingContext context = new FunctionHandlingContext(switchType)
    {
        Message = RequestHoldCall.Create(),
        DN = dn,
        Party = mainParty,
        Call = ccontext
    };

    FunctionAvailabilityPolicy policy = service.GetPolicy<FunctionAvailabilityPolicy>(context);

    Console.WriteLine(policy);
}
}

```

## Get Instructions On How To Implement a Feature

Some switches differ in how certain features can be accessed. The majority of their features may map directly to individual switch functions, but this is not always so. For example, for some switches it is not possible to log the agent out while the agent is in the ready state. So, the feature which implements agent logout for these switches would require two steps:

1. Make sure the agent is in a NotReady state
2. Log the agent out

SPL implements a feature handler for each feature that it supports. To create and run a feature handler, perform the following steps:

1. Create a new instance of FunctionHandlingContext and populate it with the following required information:
  - Specify the switch type.
  - Specify the request by setting the Message property. This step can be omitted if the feature handler is created by using the featureName parameter in the ISwitchPolicyService.CreateFeatureHandler(String featureName,



FunctionHandlingContext context) method.

- Provide a valid IProtocol instance as the value of the Protocol property.
  - Describe the context as fully as possible.
2. Call the ISwitchPolicyService.CreateFeatureHandler and pass this FunctionHandlingContext, either alone or with the name of the feature.
  3. Call the BeginExecute method on the returned handler, passing the same instance of FunctionHandlingContext.
  4. The remainder of the processing depends on the implementation, but the general approach is to perform the following actions while the status of the handler is Executing:
    1. Receive event from TServer
    2. Update FunctionHandlingContext based on the received event
    3. Assign the received event to the Message property of your FunctionHandlingContext instance
    4. Call the Handle method of IFeatureHandler passing with it the updated FunctionHandlingContext

The following code snippet shows how to do this:

```
private static void LoginReadyAgent(IProtocol protocol,
    ISwitchPolicyService service, string thisdn, string agentID)
{
    FunctionHandlingContext context = new FunctionHandlingContext("SomeSwitch");
    RequestAgentLogin requestAgentLogin = RequestAgentLogin.Create();
    requestAgentLogin.ThisDN = thisdn;
    requestAgentLogin.AgentID = agentID;
    requestAgentLogin.AgentWorkMode = AgentWorkMode.AutoIn;
    context.Message = requestAgentLogin;
    context.Protocol = protocol;

    IFeatureHandler loginHandler = service.CreateFeatureHandler(context);

    if(loginHandler==null)
    {
        protocol.Send(requestAgentLogin);
        // Process the incoming events for the scenario
        return;
    }

    // Processing feature handler
    loginHandler.BeginExecute(context);
    while (loginHandler.Status == FeatureStatus.Executing)
    {
        context.Message = context.Protocol.Receive();
        // Update the context based on the received T-Server event
        loginHandler.Handle(context);
    }
}
```

## Get Instructions On How To Accomplish Complex Functionality

Your application may sometimes need access to functionality that depends on the switch type. For example, when an application receives events from the T-Server, the way a given event's fields are used can depend on both the call scenario and the switch type. To retrieve this information, perform the following steps:

1. Create a `MessageHandlingContext` and populate it with the following required information:
  - Name of switch
  - Name of handler
2. Call `ISwitchPolicyService.CreateMessageHandler`, pass this context into it, and receive the resulting `IMessageHandler`.
3. Call the `IMessageHandler.Handle` method on the received handler.

The following code snippet shows how to do this:

```
private static void DemonstrateMessageHandler(ISwitchPolicyService service)
{
    EventRinging message = EventRinging.Create();
    message.ThirdPartyDN = "12345";
    message.DNIS = "18009870987";
    message.CallType = CallType.Internal;
    message.OtherDN = "9875";
    MessageHandlingContext context35 =
        new MessageHandlingContext("AlcatelA4400DHS3::Classic")
        { HandlerName = "OtherDN" };
    IMessageHandler handler = service.CreateMessageHandler(context35);
    string res = (string)handler.Handle(message);
    DisplayInColor(res, ConsoleColor.Yellow);
}
```

## Add Logging Support

To add logging support, carry out the following steps:

1. Create an instance of `IUnityContainer` and register an anonymous instance or type mapping for the `ILogger` interface.
2. Pass the `IUnityContainer` created during the previous step to the factory method, which creates an instance of `ISwitchPolicyService`.

The following code snippet shows how to do this:

```
IUnityContainer root = new UnityContainer();
root.RegisterInstance(new ConsoleLogger());
ISwitchPolicyService service =
    SwitchPolicyFactory.CreateSwitchPolicyService(root);
```

SPL also provides the following options:

- Your application can log the topmost messages into a distinct log. To use this option, call the `CreateSwitchPolicyService(IUnityContainer container, ILogger logger)` method of the `SwitchPolicyServiceFactory` class. The passed logger (if it is not null) will be used for logging the topmost messages.
- You can configure any switch container to use a specific logger. Objects created by the Unity container (feature handlers, policy providers and so on) can use the container to resolve the `ILogger` for further logging.

**Note:** the classes provided by SPL resolve the ILogger (if there is one) at creation time. So, if your application changes the ILogger resolution rule for the root container that was previously passed into the SwitchPolicyService constructor after the corresponding method call, this will not affect:

- Existing instances
- Objects which are created in the container(s), for which special ILogger mapping rule is configured

## Supported Functions

As mentioned above, SPL is driven by a configuration file that makes it possible to support a wide variety of switch functions. Table 1 shows the functions that are supported by SPL at installation time, using the default configuration file.

### Switch Functions Supported by SPL At Installation Time

Switch Function	Description
<b>DN and Agent Functions</b>	
RequestAgentLogin	Logs in the agent specified by the AgentId parameter to the ACD group specified by the parameter.
RequestAgentLogout	Logs the agent out of the ACD group specified by the Queue parameter.
RequestAgentNotReady	Sets a state in which the agent is not ready to receive calls. The agents telephone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestAgentReady	Sets a state in which the agent is ready to receive calls. The agents phone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestCallForwardCancel	Sets the Forwarding feature to Off for the telephony object that is specified by the DN parameter.
RequestCallForwardSet	Sets the Forwarding feature to On for the telephony object that is specified by the DN parameter.
RequestCancelMonitoring	A request by a supervisor to cancel monitoring the calls delivered to the agent. If this request is successful, T-Server distributes EventMonitoringCancelled to all clients registered on the supervisor's and agent's DNs.
RequestMonitorNextCall	A request by a supervisor to monitor (be automatically conferenced in as a party on) the next call delivered to an agent. Supervisors can request to monitor one subsequent call or all calls until the request is explicitly canceled. If a request is successful, EventMonitoringNextCall is distributed to all clients registered on the supervisor's and agent's DNs. Supervisors start monitoring each call in Mute mode. To speak, they must execute the function
RequestSetDNDOff	Sets the Do-Not-Disturb (DND) feature to Off for the telephony object specified by the DN parameter.
RequestSetDNDOOn	Sets the Do-Not-Disturb (DND) feature to On for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
<b>Call Handling Functions</b>	
RequestAlternateCall	On behalf of the telephony object specified by the DN parameter, places the active call specified by thecurrent_conn_id parameter on hold and connects the call specified by the held_conn_id parameter.
RequestAnswerCall	Answers the alerting call specified by the conn_id parameter.
RequestAttachUserData	On behalf of the telephony object specified by the DN parameter, attaches the user data structure specified by the user_data parameter to the T-Server information that is related to the call specified by the conn_id parameter.
RequestClearCall	Deletes all parties, that is, all telephony objects, from the call specified by conn_id and disconnects the call.
RequestCompleteConference	Completes a previously-initiated conference by merging the held call specified by the held_conn_id parameter with the active consultation call specified by the current_conn_id parameter on behalf of the telephony object specified by the DN. Assigns the held_conn_id to the resulting conference call. Clears the consultation call specified by the current_conn_id parameter.

RequestCompleteTransfer	On behalf of the telephony object specified by the DN parameter, completes a previously initiated two-step transfer by merging the held call specified by the conn_id parameter with the active consultation call specified by the current_conn_id parameter. Assigns held_conn_id to the resulting call. Releases the telephony object specified by the DN parameter from both calls and clears the consultation call specified by the current_conn_id parameter.
RequestDeleteFromConference	A telephony object specified by DN deletes the telephony object specified by dn_to_drop from the conference call specified by conn_id. The client that invokes this service must be a party on the call in question.
RequestDeletePair	On behalf of the telephony object specified by the DN parameter, deletes the key-value pair specified by the key parameter from the user data attached to the call specified by the conn_id parameter.
RequestDeleteUserData	On behalf of the telephony object specified by the DN parameter, deletes all of the user data attached to the call specified by the conn_id parameter.
RequestHoldCall	On behalf of the telephony object specified by the DN parameter, places the call specified by the conn_id parameter on hold.
RequestInitiateConference	On behalf of the telephony object specified by the DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the destination parameter with the purpose of a conference call.
RequestInitiateTransfer	On behalf of the telephony object specified by the DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the destination parameter for the purpose of a two-step transfer.
RequestListenDisconnect	On an existing conference call, sets Deaf mode for the party specified by the listener_dn parameter. For example, if two agents wish to consult privately, the subscriber may temporarily be placed in Deaf mode.
RequestListenReconnect	On an existing conference call, cancels Deaf mode for the party defined by the listener_dn parameter.
RequestMakeCall	Originates a regular call from the telephony object specified by the DN parameter to the called party specified by the Destination parameter.
RequestMakePredictiveCall	Makes a predictive call from the thisDN DN to the otherDN called party. A predictive call occurs before any agent-subscriber interaction is created. For example, if a fax machine answers the call, no agent connection occurs. The agent connection occurs only if there is an actual subscriber available on line.
RequestMergeCalls	On behalf of the telephony object specified by the DN parameter, merges the held call specified by the held_conn_id parameter with the active call specified by the current_conn_id parameter in a manner specified by the merge_type parameter. The resulting call will have the same conn_id as the held call.
RequestMuteTransfer	Initiates a transfer of the call specified by the conn_id parameter from the telephony object specified by the DN parameter to the party specified by the destination parameter; completes the transfer without waiting for the destination party to pick it up. Releases the telephony object specified by the DN parameter from the call.
RequestQueryCall	Requests the information specified by info_type about the telephony object specified by conn_id. If the query type is supported, the requested information will be returned in EventPartyInfo.
RequestReconnectCall	Releases the telephony object specified by the DN parameter from the active call specified by the current_conn_id parameter and retrieves the previously held call, specified by the held_conn_id parameter, to the same object. This function is commonly used to clear an active call and to return to a held call, or to cancel a consult call (due to lack of an answer, because the device is busy, and so on) and then to return to a held call.
RequestRedirectCall	Requests that the call be redirected, without an answer, from the party specified by the DN parameter to the party specified by the dest_dn parameter.
RequestRegisterAddress	Registers for a DN. Your application must register the DN before sending the RequestAgentLogin.
RequestReleaseCall	Releases the telephony object specified by the DN parameter from the call specified by the conn_id parameter.
RequestRetrieveCall	Connects the held call specified by the conn_id parameter to the telephony object specified by the DN parameter.
RequestSendDtmf	On behalf of the telephony object specified by the DN parameter, sends the digits that are expected by an interactive voice response system.
RequestSetCallInfo	Changes the call attributes. Warning: Improper use of this function may result in unpredictable behavior on the part of the T-Server and the Genesys Framework. If you have any doubt on how to use it, please consult with Genesys.

---

RequestSetMessageWaitingOff	Sets the Message Waiting indication to off for the telephony object specified by the DN parameter.
RequestSetMessageWaitingOn	Sets the Message Waiting indication to on for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
RequestSingleStepConference	Adds a new party to an existing call and creates a conference.
RequestSingleStepTransfer	Transfers the call from a specified directory number DN that is currently engaged in the call specified by the conn_id parameter to a destination DN that is specified by the destination parameter.
RequestUnregisterAddress	Unregisters a DN.
RequestUpdateUserData	On behalf of the telephony object specified by the DN parameter, updates the user data that is attached to the call specified by the conn_id parameter with the data specified by the user_data parameter.

---

# Article Sources and Contributors

**Platform SDK Developer's Guide** *Source:* <http://docs.genesyslab.com/wiki/index.php?oldid=3547> *Contributors:* Edjamer, WikiSysop

**Developers Guide PDF - PSDK 7.6** *Source:* <http://docs.genesyslab.com/wiki/index.php?oldid=3542> *Contributors:* Edjamer, WikiSysop

**LCA Hang-Up Detection Support** *Source:* <http://docs.genesyslab.com/wiki/index.php?oldid=3544> *Contributors:* Edjamer, WikiSysop

**Lazy Parsing of Message Attributes** *Source:* <http://docs.genesyslab.com/wiki/index.php?oldid=3545> *Contributors:* Edjamer, WikiSysop

**Using the Switch Policy Library** *Source:* <http://docs.genesyslab.com/wiki/index.php?oldid=3550> *Contributors:* Edjamer, WikiSysop

# Image Sources, Licenses and Contributors

**File:welcome.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:Welcome.png> License: unknown Contributors: WikiSysop  
**File:DevGuide2.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:DevGuide2.png> License: unknown Contributors: WikiSysop  
**File:Important.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:Important.png> License: unknown Contributors: WikiSysop  
**File:PDF.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:PDF.png> License: unknown Contributors: WikiSysop  
**File:Download.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:Download.png> License: unknown Contributors: WikiSysop  
**Image:Welcome.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:Welcome.png> License: unknown Contributors: WikiSysop  
**Image:DevGuide2.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:DevGuide2.png> License: unknown Contributors: WikiSysop  
**File:Information.png** Source: <http://docs.genesyslab.com/wiki/index.php?title=File:Information.png> License: unknown Contributors: WikiSysop