



CHAPTER 2

General API Concepts

This module provides a description of the various concepts that are used when working with the SM C/C++ API.

- [Information About the Blocking API and the Nonblocking API, page 2-1](#)
- [Reliability, page 2-2](#)
- [C versus C++ API, page 2-2](#)
- [Information About API Initialization, page 2-4](#)
- [API Finalization, page 2-5](#)
- [Information About the Return Code Structure, page 2-6](#)
- [Information About the Error Code Structure, page 2-7](#)
- [Subscriber Name Format, page 2-7](#)
- [Information About Network ID Mappings, page 2-8](#)
- [Subscriber Domains, page 2-10](#)
- [Subscriber Properties, page 2-10](#)
- [Custom Properties, page 2-10](#)
- [Logging Capabilities, page 2-11](#)
- [Information About the Disconnect Callback Listener, page 2-11](#)
- [Signal Handling, page 2-11](#)
- [Practical Tips, page 2-12](#)

Information About the Blocking API and the Nonblocking API

This section describes the differences between the Blocking API and Nonblocking API operations.

- [Blocking API, page 2-2](#)
- [Nonblocking API, page 2-2](#)

Blocking API

In a Blocking API operation, which is the common type, every method returns *after* its operation has been performed.

The SM Blocking C/C++ API provides a wide range of operations and is a *superset* of the Nonblocking API functionality.

Nonblocking API

In a Nonblocking API operation, every method returns *immediately*, even before the completion of its operation. The operation results are returned either to a set of user defined callbacks or not returned at all.

The Nonblocking method is advantageous when the operation is lengthy and involves I/O. Performing the operation in a separate thread allows the calling program to continue doing other tasks and it improves overall system performance.

The SM Nonblocking C/C++ API contains a small number of Nonblocking operations. The API supports retrieval of operation results using result callbacks.

Reliability

The SCMS SM C/C++ API is reliable in the sense that the operations performed on the API are kept until the SM returns their associated results. If the connection to the SM is lost, operations that were not sent to the SM and operations whose results did not yet return from the SM are sent to the SM immediately on reconnection. The order of operation invocation is maintained at all times.

C versus C++ API

The C and C++ APIs are essentially the same. The C API is actually a thin wrapper over the C++ API, with method prototype and signature changes imposed by the constraint of C not being an object-oriented programming language.

The following sections describe the C/C++ API differences and provide some examples.

- [Method Names, page 2-2](#)
- [Handle Pointers, page 2-3](#)

Method Names

The method names of the C API are the same as in the C++ API, except that the C API method names have an identifying prefix:

- The Blocking C API method names are prefixed with SMB_.
- The Nonblocking C API methods are prefixed with SMNB_.



Note

The documentation of methods in this guide uses the name and signature of the C++ API methods.

For example:

Both the Blocking and Nonblocking C++ APIs provide a **login** method. The method in the Blocking C API is called **SMB_login** and the method in the Nonblocking C API is called **SMNB_login**.

Handle Pointers

The Blocking and Nonblocking APIs are C++ Objects. Several API instances can co-exist in a single process, each having its own socket and state. To provide the same functionality in the C API, the handle concept is introduced. Each C API method accepts a handle as its first argument. The handle is used to identify the C API instance on which the caller wants to operate. Calling the **SMB_init** or **SMNB_init** method creates a new C API instance; the return value of these methods provides the handle to the instance. For more information, see [Information About API Initialization, page 2-4](#).

- [Blocking API Example, page 2-3](#)
- [Nonblocking API Example, page 2-3](#)

Blocking API Example

The following C++ Blocking API **logoutByName** method signature:

```
ReturnCode* logoutByName(char* argName,
                        char** argMappings,
                        MappingType* argMappingTypes,
                        int argMappingsSize)
```

is translated into C Blocking API as:

```
ReturnCode* SMB_logoutByName(SMB_HANDLE argApiHandle,
                            char* argName,
                            char** argMappings,
                            MappingType* argMappingTypes,
                            int argMappingsSize);
```

Nonblocking API Example

The following C++ Nonblocking API **logoutByName** method signature:

```
int logoutByName(char* argName,
                char** argMappings,
                MappingType* argMappingTypes,
                int argMappingsSize);
```

is translated into C Nonblocking API as:

```
int SMNB_logoutByName(SMNB_HANDLE argApiHandle,
                    char* argName,
                    char** argMappings,
                    MappingType* argMappingTypes,
                    int argMappingsSize);
```

Information About API Initialization

There are three main steps to initialize the API:

- Construct the API using one of its two constructors.
- Perform API-specific setup operations.
- Connect the API to the SM.

The following sections describe these three steps.

Initialization examples can be found in the code examples sections under each API.

API Construction

Both C and C++ Blocking and Nonblocking APIs must construct and initialize the API. Be sure to check that the initialization has succeeded before proceeding.

To construct and initialize the C++ API, construct an API object and call the **init** function as shown in the following example:

```
SmApiNonBlocking nbapi;
if (!nbapi.init(0,2000000,10,30)){
    exit(1);
}
```

To construct and initialize the C API, call the **init** function, which allocates and initializes the API. For example:

```
SMNB_HANDLE nbapi = SMNB_init(0,2000000,10,30);
if (nbapi == NULL){
    exit(1);
}
```

Setting the LEG Name

Set the Login Event Generator (LEG) name if you intend to turn on the SM-LEG failure handling options in the SM. For more information about LEGs and SM-LEG failure handling, see the [Cisco SCMS Subscriber Manager User Guide](#).

To set the LEG name, call the relevant **setName** function in the API. The SM will use the LEG name when recovering from a connection failure. A constant string that identifies the API will be appended to the LEG name as follows:

- For Blocking API: **.B.SM-API.C**
- For Nonblocking API: **.NB.SM-API.C**

Setting the LEG Name Example (Blocking API)

If the provided LEG name is **my-leg.10.1.12.45-version-1.0**, the actual LEG name will be **my-leg.10.1.12.45-version-1.0.B.SM-API.C**.

If no name is set, the LEG uses the hostname of the machine as the prefix of the name.

For additional information about SM-LEG failure handling, see [Appendix A](#) of the *Cisco SCMS Subscriber Manager User Guide*.

Setup Operations

The setup operations for the two APIs differ. Both APIs support setting a disconnect listener, which is described in [Information About the Disconnect Callback Listener, page 2-11](#).

The following sections describe the setup operations for the Blocking API and the Nonblocking API.

- [Blocking API Setup, page 2-5](#)
- [Nonblocking API Setup, page 2-5](#)

Blocking API Setup

To set up the Blocking API, you must set an operation timeout value. For more information, see the [Blocking API](#) module.

Nonblocking API Setup

To set up the Nonblocking API you are required to set a disconnect callback, see the [Nonblocking API](#) module.

Connecting to the Subscriber Manager

In order to connect to the SM, use one of the following connect methods:

- The following example shows how to connect when using the C++ APIs:

```
connect(char* host, Uint16 argPort = 14374)
```

- The following example shows how to connect when using the C Blocking API:

```
SMB_connect(SMB_HANDLE argApiHandle, char* host, Uint16 argPort)
```

- The following example shows how to connect when using the C Nonblocking API:

```
SMNB_connect(SMNB_HANDLE argApiHandle, char* host, Uint16 argPort)
```

The **argHostName** parameter can be either an IP address or a reachable hostname. At any time during the API operation, you can check if the API is connected by using one of the variants of the function **isConnected**.

API Finalization

Both C and C++ Blocking and Nonblocking APIs must disconnect from the SM and free the memory of the API:

- For the C++ APIs, call the **disconnect** method and free the API object.
- For the C APIs, call the appropriate **disconnect** function and then free the API using the appropriate **release** function.

Information About the Return Code Structure

The results of the API operations are returned using a generic structure called **ReturnCode**. The **ReturnCode** structure contains several parameters:

- **u**—A union of all of variables and pointers to variables that are the actual returned value.
- **type**—A return code type parameter (**ReturnCodeType** enumeration) that defines the type of the value (**u**) that the **ReturnCode** structure holds.
- **size**—The number of elements in the value. If **size** equals 1 then there is one element in the value, such as a scalar, character string, void, or error. If **size** is greater than 1 then there are several elements in the array, and the type should be one of the array types.

The API allocates the return code structure and the API user must free the structure. You can use the **freeReturnCode** utility function to safely free the structure.

Additional return code structure utility functions are:

- **printReturnCode**—Prints the **ReturnCode** structure value to the stdout
- **isReturnCodeError**—Checks whether the **ReturnCode** structure is an error

Definitions

From the **GeneralDefs.h** header file:

```
OSAL_DllExport typedef struct ReturnCode_t
{
    ReturnCodeType type;
    int size;          /* number of elements in the union element */
                    /* (for example: stringVal will have size=1) */
    union { /* use union value according to the type value */
        bool boolVal;
        short shortVal;
        int intVal;
        long longVal;
        char* stringVal;
        bool* boolArrayVal;
        short* shortArrayVal;
        int* intArrayVal;
        long* longArrayVal;
        char** stringArrayVal;
        ErrorCode* errorCode;
        struct ReturnCode_t** objectArray;
    } u;
}ReturnCode;
```

Return Code Structure Example

In the following example, the subscriber data of *subscriber1* is retrieved and displayed. The returned structure contains an array of `ReturnCode` structures held by the **objectArray** union value. Each structure contains a different value type. For additional information, see the explanation of the [getSubscriber](#) method. This example code uses the **isReturnCodeError** and **freeReturnCode** methods.

```
ReturnCode* subFields = bapi.getSubscriber("subscriber1");
if (isReturnCodeError(subFields) == false)
{
    printf("\tname:\t\t%s\n", subFields->u.objectArray[0]->u.stringVal);
    printf("\tmapping:\t%s\n", subFields->u.objectArray[1]->u.stringArrayVal[0]);
    printf("\tdomain:\t\t%s\n", subFields->u.objectArray[3]->u.stringVal);
    printf("\tautologout:\t%d\n", subFields->u.objectArray[8]->u.intVal);
}
else
{
    printf("error in subscriber retrieval\n");
}
freeReturnCode(subFields);
```

Information About the Error Code Structure

The **ErrorCode** structure can be one of the values of the return code structure. This structure describes the error that occurred. The structure consists of the following parameters:

- **type**—An `ErrorCodeType` enumeration that describes the error. See the **GeneralDefs.h** file for additional information
- **message**—A specific error code
- **name**—Currently not used

Error Code Structure Definition

From the **GeneralDefs.h** header file:

```
OSAL_DllExport typedef struct ErrorCode_t
{
    ErrorCodeType type; /* type of the error see enumeration */
    char* name; /* currently not used */
    char* message; /* error message */
}ErrorCode;
```

Subscriber Name Format

Most methods of both APIs require the subscriber name as an input parameter. This section lists the formatting rules of a subscriber name.

It can contain up to 64 characters. All printable characters with an ASCII code between 32 and 126 (inclusive) can be used; except for 34 ("), 39 ('), and 96 (`).

Information About Network ID Mappings

A network ID mapping is a network identifier that the SCE device can relate to a specific subscriber record. A typical example of a network ID mapping (or simply mapping) is an IP address. For additional information, see the *Cisco SCMS Subscriber Manager User Guide*. Currently, the Cisco Service Control Solution supports IP address, IP range, private IP address over VPN, private IP range over VPN, and VLAN mappings.

Both Blocking and Nonblocking APIs contain operations that accept mappings as a parameter. Examples are:

- The **addSubscriber** operation (Blocking APIs)
- The **login** method (Blocking or Nonblocking APIs)

When passing mappings to an API method, the caller is requested to provide two parameters:

- A character string (**char***) mapping identifiers or array of character strings (**char****) mappings.
- A **MappingType** enumeration or array of **MappingType** variables.

When passing arrays, the **MappingType** variables array must contain the same number of elements as the mappings array.

The API supports the following subscriber mapping types (defined by the **MappingType** enumeration):

- IP addresses or IP ranges
- Private IP addresses or private IP ranges over VPN
- VLAN tags

Specifying IP Address Mapping

The string format of an IP address is the commonly used decimal notation:

```
IP-Address=[0-255].[0-255].[0-255].[0-255]
```

Example:

- 216.109.118.66

The **GeneralDefs.h** header file provides the mapping type of an IP address:

- **IP_RANGE** specifies IP mapping (IP-Address or IP-Range that matches the mapping identifier with the same index in the mapping identifier array).

Specifying IP Range Mapping

The string format of an IP range is an IP address in decimal notation and a decimal specifying the number of 1s in a bit mask:

IP-Range=[0-255] . [0-255] . [0-255] . [0-255] / [0-32]

Examples:

- **10.1.1.10/32** is an IP range with a full mask, that is, a regular IP address.
- **10.1.1.0/24** is an IP range with a 24-bit mask, that is, all the addresses ranging between **10.1.1.0** and **10.1.1.255**.



Note

The mapping type of an IP Range is identical to the mapping type of the IP address.

Specifying Private IP Address or Private IP Range over VPN Mapping

The string format of an IP address and an IP range are described in [Specifying IP Address Mapping](#) and [Specifying IP Range Mapping](#). When the network ID mapping uses an IP address or range over VPN, the string format includes the VPN name.

Examples:

- **10.1.1.10@VPN1** is an IP address over the VPN named *VPN1*.
- **10.1.1.0/24@VPN2** is an IP range with a 24-bit mask, that is, all of the addresses ranging between **10.1.1.0** and **10.1.1.255** over the VPN named *VPN2*.



Note

The mapping type of an IP address or IP range over VPN is identical to the mapping type of the IP address.

Specifying VLAN Tag Mapping

The string format for VLAN tag mapping is: **VLAN-tag = 0-4095**.

The string is a decimal in the specified range.

The **GeneralDefs.h** header file provides the mapping type of a VPN which is used to specify VLAN-tags as network IDs.



Note

The mapping type VLAN is deprecated and it is recommended to use the mapping type VPN instead.

Subscriber Domains

The *Cisco SCMS Subscriber Manager User Guide* explains the domain concept in detail. Briefly, a domain is an identifier that tells the SM which SCE devices to update with the subscriber record.

A domain name is of type (`char*`). During system installation, the network administrator determines the system domain names, which therefore vary between installations. The APIs include methods that specify to which domain a subscriber belongs and allow queries about the system's domain names. If an API operation specifies a domain name that does not exist in the SM domain repository, it is considered an error and an **ERROR_CODE_DOMAIN_NOT_FOUND** error **ReturnCode** will be returned.

The SM's *Automatic Domain Roaming* feature allows subscribers to be moved between domains by calling the `login` method for a subscriber with an updated domain parameter.



Note

Automatic domain roaming is not backwards compatible with previous versions of the SM API, which did not allow changing the domain of the subscriber.

Subscriber Properties

Several operations manipulate subscriber properties. A subscriber property is a key-value pair that affects the way the SCE analyzes and reacts to network traffic generated by the subscriber.

More information about properties can be found in the *Cisco SCMS Subscriber Manager User Guide* and in the *Cisco Service Control Application for Broadband User Guide*. The application user guide provides application-specific information. It lists the subscriber properties that exist in the application running on your system, the allowed value set, and the significance of each property value.

To format subscriber properties for C/C++ API operations, use the String arrays (`char**`) **propertyKeys** and **propertyValues**.



Note

The arrays must be of the *same length*, and NULL entries are forbidden. Each key in the keys array has a matching entry in the values array. The value for **propertyKeys[j]** resides in **propertyValues[j]**.

Example:

If the property keys array is {`"packageId","monitor"`} and the property values array is {`"5","1"`}, the properties will be **packageId=5, monitor=1**.

Custom Properties

Some operations manipulate custom properties. Custom properties are similar to subscriber properties, but do not affect how the SCE analyzes and manipulates the subscriber's traffic. The application management modules use custom properties to store additional information for each subscriber.

To format custom properties, use the string (`char**`) arrays **customPropertyKeys** and **customPropertyValues**, the same as used in formatting [Subscriber Properties](#).

Logging Capabilities

The API package contains a **Logger** abstract class that can be inherited and used to integrate the SM API with the host application log. The **Logger** class exposes four basic levels of logging: error messages, warning messages, informative messages, and several levels of trace messages. Both the Blocking and Nonblocking API have this capability. The **Logger.h** header file provides the **Logger** class.

The API user should implement a logger by inheriting from the **Logger** class. To have the API use this logger, the code should call the API's **setLogger** method of the C++ implementation of the API.

For testing and for simple LEG implementations, the API package provides the **PrintLogger** class, which is a simple implementation of the **Logger** class that prints the log messages to the standard error (STDERR). An API user can initiate the **PrintLogger** object, set its logging level using the **setLoggingLevels** method of the **PrintLogger** class, and pass the logger object to the API using the API **setLogger** method. The **PrintLogger.h** header file provides the **PrintLogger** class.

Information About the Disconnect Callback Listener

Both Blocking and Nonblocking APIs allow setting a disconnect callback listener to be notified when the API is disconnected from the SM. The disconnect callback is defined as follows:

```
typedef void (*ConnectionIsDownCallBackFunc)();
```

To set the disconnect listener, use the **setDisconnectListener** method.

Disconnect Callback Listener Example

The following example is a simple implementation of a disconnect callback listener that prints a message to **stdout** and exits:

```
#include "GeneralDefs.h"
void connectionIsDown(){
    printf("Message: connection is down.");
    exit(0);
}
```

Signal Handling

The SCMS SM C/C++ API as a networking module might handle sockets that are closed by the SM, for example, if the SM is restarted, which may cause "Broken Pipe" signals. It is especially advisable for the UNIX environment to handle this signal.

To ignore the signal, add the following call:

```
sigignore(SIGPIPE);
```

Practical Tips

When implementing the code that integrates the API with your application, you should consider the following practical tips:

- Connect to the SM once and maintain an open API connection to the SM at all times, using the API many times. Establishing a connection is a timely procedure, which allocates resources on the SM side and the API client side.
- Share the API connection between your threads. It is better to have one connection per LEG. Multiple connections require more resources on the SM and client side.
- Do not implement synchronization of the calls to the API. The client automatically synchronizes calls to the API.
- It is recommended to place the API clients (LEGs) in the same order of the SM machine processor number.
- If the LEG application has bursts of logon operations, enlarge the internal buffer size accordingly to hold these bursts (Nonblocking flavor).
- During the integration, set the SM **logon_logging_enabled** configuration parameter to view the API operations in the SM log to troubleshoot the integration, if any problems arise.
- Use the debug mode for the LEG application that logs/prints the return values of the nonblocking operations.
- Use the automatic reconnect feature to improve the resiliency of the connection to the SM.
- In cluster setups, connect the API using the virtual IP address of the cluster and not the management IP address of one of the machines.