CyberSource SCMP API Client

Developer Guide

September 2019

CyberSource®
the power of payment

CyberSource Corporation HQ | P.O. Box 8999 | San Francisco, CA 94128-8999 | Phone: 800-530-9095

CyberSource Contact Information

For general information about our company, products, and services, go to http://www.cybersource.com.

For sales questions about any CyberSource Service, email sales@cybersource.com or call 650-432-7350 or 888-330-2300 (toll free in the United States).

For support information about any CyberSource Service, visit the Support Center at http://www.cybersource.com/support.

Copyright

Restricted Rights Legends

Trademarks

Contents

Recent Revisions to This Document

Release	Changes
September 2019	Deprecated and deleted Perl and ASP sections and removed ASP and Perl section references from About this Guide and Introduction sections.
September 2015	Updated the production server URL and the test server URL.
January 2015	Removed information about the Stored Value service, which is no longer supported.
August 2013	This revision contains only editorial changes and no technical updates.
May 2013	■ Noted that the SCMP API clients are supported only on 32-bit operating systems. ■ Combined all SCMP API client documents into this developer guide, which covers all supported programming languages.
January 2013	■ Noted a change in transaction security key use requirements for the CyberSource production and test environments. For more information, see the “Transaction Security Keys” section in each programming language chapter: • ASP client • C/C++ client • Java client • .NET client • Perl client ■ Removed information about how to generate security keys and added it to *Creating and Using Security Keys* (PDF | HTML).

About This Guide

Audience

This guide is written for application developers who want to use the CyberSource SCMP API client to integrate the following CyberSource services into their order management system:

- CyberSource Essentials
- CyberSource Advanced

Using the SCMP API client SDK requires programming skills in one of the supported programming languages:

- C, C++
- Java
- .NET

To use these SDKs, you must write code that uses the API request and reply fields to integrate CyberSource services into your existing order management system.

Purpose

This guide describes tasks you must complete to install, test, and use the CyberSource SCMP API client software.

Scope

This guide describes how to install, test, and use all available SCMP API clients. It does not describe how to implement CyberSource services with the SCMP API. For information about how to use the API to implement CyberSource services, see "Related Documents," page 11.

Conventions

Note and Important Statements


Note

A *Note* contains helpful suggestions or references to material not contained in this document.


Important

An *Important* statement contains information essential to successfully completing a task or learning a concept.

Text and Command Conventions

Convention	Usage
bold	■ Field and service names; for example: Include the **ics_applications** field. ■ Items that you are instructed to act upon; for example: Click **Save**.
italic	■ Filenames and pathnames. For example: Add the filter definition and mapping to your *web.xml* file. ■ Placeholder variables for which you supply particular values.
`screen text`	■ XML elements. ■ Code examples and samples. ■ Text that you enter in an API environment; for example: Set the **davService_run** field to `true`.

Related Documents

Client Package Documentation

The following documentation is available in the client package download:

- README file
- CHANGES file
- Sample code files

CyberSource Services Documentation

This guide (*SCMP API Client Developer Guide*) contains information about how to:

- Create the request
- Send the request
- Receive the reply

In contrast, CyberSource services documentation listed in Table 1 contains information about how to:

- Identify what to put in requests sent to CyberSource.
- Interpret what is contained in the reply from CyberSource.

Each type of CyberSource service has associated documentation:

Table 1 CyberSource Services Documentation

Type of Service	Available Documentation
CyberSource Essentials	■ *Credit Card Services User Guide* (PDF | HTML) ■ *Electronic Check Services User Guide* (PDF | HTML)
CyberSource Advanced	■ *Credit Card Services Using the SCMP API* (PDF | HTML) ■ *Reporting User Guide* (PDF | HTML)

If you use other CyberSource services, the documentation can be found on the CyberSource Essentials or CyberSource Advanced (Global Payment Services) sections of the CyberSource web site.

Customer Support

For support information about any CyberSource service, visit the Support Center:

http://www.cybersource.com/support

CHAPTER 1

Introduction

Important

The SCMP API clients are supported on 32-bit operating systems only.

CyberSource SCMP (Simple Commerce Message Protocol) is a legacy name-value pair API that is supported for existing implementations. If you are new to CyberSource and want to connect to CyberSource services by using an API, use the Simple Order API.

The SCMP API SDKs provide the client software for the following programming languages:

- C, C++
- .NET 2002, .NET 2003
- Java

As a user of CyberSource services, you can use the Business Center to process your customers' payments and credits, view details about your orders, and download reports. After you register your merchant ID with CyberSource, you can use the test Business Center. After you go live, you can also use the production Business Center. To log in, you need the name of your organization or merchant ID, your user name, and your password. These items were established when your company registered the merchant ID with CyberSource.

If you need help with the Business Center features, click the Online Help link.

CHAPTER 2

C/C++ Client

Important

The SCMP API clients are supported on 32-bit operating systems only.

Basic C Program Example

The following example shows the C code required to send a request for credit card authorization and to process the results. See "Using the SDK," page 21 for details on writing the code.

Example 1 Example of Credit Card Authorization Request in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ics.h"
main()
{
  /* Allocate request and result space */
  ics_msg *request;
  ics_msg *result;
  char *s = NULL;
  int rcode;

  /* Initialize the request, which holds all the information */
  /* sent to CyberSource */
  request = ics_init(0);
  if (request == NULL) {
    /* Initialization failed, do something appropriate */
  }
  /* Add your company's information to the request */
  ics_fadd(request, "merchant_id", "infodev");

  /* Add the services you want to the request */
  ics_fadd(request, "ics_applications", "ics_auth");
  ics_fadd(request, "server_host", "ics2testa.ic3.com");
  /* Add the customer information to the request */
```

```
ics_fadd(request, "customer_firstname", "John");
ics_fadd(request, "customer_lastname", "Doe");
ics_fadd(request, "customer_email", "nobody@cybersource.com");

ics_fadd(request, "customer_phone", "6509656000");
/* set more fields as necessary */
/* (not all required fields shown here) */

/* Add the offer information to the request */
ics_fadd(request, "offer0", "amount:10.99^quantity:1");

/* send the request to CyberSource server */
result = ics_send(request);

/* Handle the reply codes and error messages. */
if (result == NULL) {
  printf("ics error: got a NULL reply.\n");
}
s = ics_fgetbyname(result, "ics_rcode");
if (s == NULL) {
  rcode = -1;
} else {
  rcode = atoi(s);
}
if (rcode < 0) {
  printf ("ics error: %s\n", ics_fgetbyname(result, "ics_rmsg"));
  /* Error occurred */
  /* Notify customer that error occurred; */
  /* ask customer to try again. */
}
else if (rcode == 0) {
  /* Request declined */
  /* Evaluate rflag for reason why declined */
  s = ics_fgetbyname(result, "ics_rflag");
  if (strcmp(s, "DINVALIDDATA") == 0) {
    /* Notify customer that request included invalid data. */
  }
  else if (strcmp(s, "DCARDREFUSED") == 0) {
    /* Card declined by the bank; */
    /* notify customer that unable to process order. */
  }
  else {
    /* Handle remaining possible declined reply flags here. */
    /* Write error handler to process */
    /* new reply flags created by CyberSource. */
  }
}

else {
  /* Successful transaction; */
  /* perform any local processing, and */
  /* notify customer of success. */
}
```

```
  ics_destroy (request);
  ics_destroy (result);
}
```

Installing and Configuring the SDK

To install the SDK, you must first download the SDK file and then run an installation program included in the package. The installation procedure varies depending on if you are a new customer or if you are upgrading the SDK from a previous version.

Minimum System Requirements

- One of these operating systems:
  - Linux kernel 2.2, libc 6 with Intel
  - Solaris 2.6 operating system on SPARC
  - Windows NT 4.0 with Intel
- 20 MB of disk space
- A utility to uncompress the distribution file

The SDK supports ISO-8859-1 encoding.

Important

Failure to configure your client API host to a unique, public IP address causes inconsistent transaction results.

The client API request ID algorithm uses a combination of IP address and system time, along with other values. In some architectures this combination might not yield unique identifiers. If it is not possible to configure your machines with unique IP addresses, CyberSource provides a client configuration parameter that you can use to identify your host. For the CyberSource SCMP API SDK for C/C++, this configuration parameter is the ***ics.host_id*** entry in the configuration file. The value must not be 127.0.0.1, which is the loopback address of your localhost.

Downloading the SDK

To download the SDK:

Step 1 Create a target directory.

Step 2 Download the latest SDK package file from the downloads area of the CyberSource Support Center.

Step 3 Unzip/untar the package to your target directory.

Important

Unzipping and untarring the original SDK package installs several directories and files in your target directory. The installation program then creates a new directory and copies most of the files from your target directory into this new directory. However, the following files are not copied and exist only in your target directory: `README.txt`, `CHANGES.txt`, and `LICENSE.txt`. Copy the files to the new directory where the SDK is installed so that you can later delete your target directory.

Installing the SDK for the First Time

If you are a new customer and do not have a previous version of the C/C++ SDK installed, simply run the installation program.

Linux and Solaris

Type **`sh install.sh`**

The default installation path is **`/opt/CyberSource/SDK`**.

To install the SDK to a different directory, enter the path when prompted during the installation.

Note

To install the SDK to an alternate directory, you must set the ICSPATH environment variable to indicate the alternate location of the SDK. For example, set ICSPATH to `/opt/ics`.

Windows

Type this with your CyberSource merchant ID:

```
install.bat myMerchantID
```

The default installation path is `c:\opt\CyberSource\SDK\`.

To install the SDK to a different directory, type

```
install.bat myMerchantID path
```

If you install the SDK to an alternate directory, you must set the ICSPATH environment variable to indicate the alternate location of the SDK. For example, set ICSPATH to `\opt\ics`.

Indicating the Path to Your Security Keys

You must generate two transaction security keys—one for the CyberSource production environment and one for the test environment. For information about generating and using security keys, see *Creating and Using Security Keys* (PDF | HTML).

The client looks for the certificate and private key files in `/opt/CyberSource/SDK/keys` by default. To store your keys in a different directory, set an `ICSPATH` environment variable. The client automatically adds `/keys` and looks for the keys in the `$ICSPATH/keys` location. For example, when you set `ICSPATH` to *`myDirectory`*, the client looks for the keys in *`myDirectory`*`/keys` folder.

Upgrading the SDK

To upgrade an existing SDK installation:

Step 1 Rename the directory containing your previous SDK installation.

For example, rename `opt/CyberSource/SDK to opt/CyberSource/oldSDK`.

Step 2 Run the installation program:

- **Linux and Solaris**: type `sh install.sh`

  The default installation path is `/opt/CyberSource/SDK`. To install the SDK to a different directory, you can enter the path when prompted during the installation.

To install the SDK to an alternate directory, you must set the ICSPATH environment variable to indicate the alternate location of the SDK. For example, set ICSPATH to `/opt/ics`.

- **Windows**: type (including your merchant ID) `install.bat` ***`myMerchantID`***

  The default installation path is `c:\opt\CyberSource\SDK\`. To install the SDK to a different directory, type `install.bat` ***`myMerchantID`*** `path`.

To install the SDK to an alternate directory, you must set the ICSPATH environment variable to indicate the alternate location of the SDK. For example, set ICSPATH to `\opt\ics`.

Step 3 Copy your original key files from the old `keys` directory to the new `keys` directory (overwriting any files in the new `keys` directory).

For example, copy all the files from `/CyberSource/oldSDK/keys/` to `opt/CyberSource/SDK/keys/`.

Step 4 When you are confident that the new SDK is working, delete the `oldSDK` directory.

Parts of the SDK

The C/C++ SDK includes the directories and files listed in this table.

Directory or file	Description
`bin`	Contains the ECert program that creates your certificate and private keys. Transaction security keys can also be generated in the Business Center. For more information, see *Creating and Using Security Keys* (PDF | HTML).
`include`	Contains the header file `ics.h`.
`keys`	Contains your certificate and private keys.
`lib`	Contains the C libraries.
`sample`	Contains test programs (in the `icstest` directory) and sample requests (in the `requests` directory). `icstest`: Runs a sample transaction. Use to test the SDK installation. `mt_test`: Reads request data from a text file and sends the request. Troubleshoots problems with sending requests. Use with the sample requests or your own requests, for example: `mt_test ../requests/ics_auth.txt`
`CHANGES.txt`	Describes the changes in this and previous versions of the SDK.
`LICENSE.txt`	Contains CyberSource software license information.
`README.txt`	Contains information about SDK installation, shared libraries, C++ support, supported platforms, and test transactions.
`install.sh`	Sets up the SDK file structure, installs the libraries, and runs the ECert application.

Creating the Configuration File

You can create a configuration file that stores fields that you need for every request. Use the configuration file as an alternative to specifying the values for these fields in the request.

You can give the file any name and store it in any directory. Specify the file name and path by using the `ics_set_config_file()` function. The function tells the request to look for a configuration file to retrieve these particular field values. See "ics_set_config_file() Function," page 36 for more information.

These are the fields that you can include in the configuration file:

- `debug` — enables debug output. See "ics_init() Function," page 35 for more information about debugging.
- `ics.host_id` — local IP address to use for generating a request ID
- `ics.http_proxy` — URL for CyberSource HTTP proxy
- `ics.http_proxy_password` — your password for CyberSource HTTP proxy
- `ics.http_proxy_username` — your username for CyberSource HTTP proxy
- `ics.merchant_id` — your merchant ID
- `ics.server_host` — default CyberSource server name
- `ics.server_port` — default CyberSource server port

To create a configuration file, create a text file that includes the configuration parameters in a name-value pair format, for example:

```
debug=0
ics.merchant_id=MyMerchantID
ics.server_host=ics2a.ic3.com
ics.server_port=80
```

Testing the SDK Installation

To test the SDK installation:

Step 1 Open a command prompt.

Step 2 Go to the `sample/icstest` directory in the SDK.

Step 3 Run the test program:

For Linux and Solaris, type `./icstest` ***`merchant_id`***

For Windows, type `icstest` ***`merchant_id`***

If you successfully installed and configured the SDK, the following message appears: `successful ICS order, rcode = 1`.

If you do not receive this message, verify that your ICSPATH variable is set correctly, and that your key files are in the correct location.

Using the mt_test Program

You may also create and send your own requests by using the `mt_test` program. The SDK contains several sample requests that you can use with the program (see the `sample/requests` directory).


Note

If you provided a merchant ID when you installed the SDK, the request text files were automatically updated with your merchant ID. If you did not provide a merchant ID during installation, edit the line in the `ics_auth.txt` file that sets your merchant ID.

When in test mode, send your requests to the test server ics2test.ic3.com and use port 80. See the sample requests for examples.

To use the mt_test program with the ics_auth.txt sample request:

Step 1 Open a command prompt.

Step 2 In the directory where you installed the SDK, go to the `sample/icstest` directory.

Step 3 Type `./mt_test ..requests/ics_auth.txt`

The test results appear in the window.

Going Live

When you complete all of your system testing and are ready to accept real transactions from your customers, your deployment is ready to *go live*. When your deployment goes live, you can send transactions to CyberSource's production server. Provide your banking information to CyberSource so that your processor can deposit funds to your merchant bank account.

To send transactions to the CyberSource production server:

Step 1 Log in to the test Business Center.

Step 2 In the left navigation panel of the test Business Center, click **Support Center**.

Step 3 On the Support Center home page, in the **Search Knowledgebase** search well, type **go live**, and then click **Search**.

Step 4 In the search results, depending on what version of CyberSource you use, click **How do I go live? (Enterprise)** or **How do I go live? (Small Business)**.

These knowledgebase articles provide instructions that describe how to activate your account to process live transactions.

Once CyberSource has confirmed that your deployment is live, make sure to update your system so that you send requests to the production server (`ics2a.ic3.com`) instead of the test server (`ics2testa.ic3.com`).

After your deployment goes live, use real card numbers and other data to test every card type, currency, and CyberSource application that your integration supports. Because these are real transactions, use small monetary amounts to do the tests. Process an authorization, capture, and credit for each configuration. Use your bank statements to verify that money is deposited into and withdrawn from your merchant bank account. If you have more than one CyberSource merchant ID, test each one separately.

Using the SDK

This section explains how to request CyberSource services using the C/C++ SDK.

Requesting CyberSource Services

To request CyberSource services, you must create these items:

- A system that collects the information required for the CyberSource services
- C/C++ programs to assemble the order information into request, send the request to the CyberSource server, and process the reply information

The CyberSource services use the SCMP API, which consists of name-value pairs. The name-value pair API fields you use for credit card orders are described in the *Credit Card Services Using the SCMP API* (PDF | HTML). The instructions in this section show you how to write C/C++ programs that correctly use SCMP API name-value pairs.

The code in the examples in this section is incomplete. For complete sample programs, see the test programs in the `sample/icstest` directory of the SDK.

Constructing and Sending Requests

To access any CyberSource services, you must create and send a request that holds the required information. The example in this section shows basic code for requesting CyberSource services. In this example, John Doe is buying three different items, and you are requesting credit card authorization. For the list and description of all the CyberSource C functions, see "C Functions," page 31.

Adding Header Files and Variables

Include the header file and declare the request and result variables:

```
#include "ics.h"
ics_msg *icsorder;
ics_msg *res;
```

Creating the Request

Next, create a new request:

```
icsorder = ics_init(0);
```

Adding Services to the Request

Next, add the service that you want using the `ics_fadd()` function:

```
ics_fadd(icsorder, "ics_applications", "ics_auth");
```

You can request multiple services by using commas to separate the service names. For example, you can request both credit card authorization and capture together, a sale, by using the following code:

```
ics_fadd(icsorder, "ics_applications", "ics_auth,ics_bill");
```

Adding Request-Level Fields

Next, add request-level fields using the `ics_fadd()` function. The fields include basic information about the customer and your company. You can include embedded spaces in the values, but if you include multiple consecutive spaces, only one of the spaces is retained when the request is processed.

```
ics_fadd(icsorder, "merchant_id", merchant_id);
ics_fadd(icsorder, "customer_firstname", "John");
ics_fadd(icsorder, "customer_lastname", "Doe");
ics_fadd(icsorder, "customer_cc_number", "4111111111111111");
```

The example above shows only a partial list of the fields required for the request.

Adding Offer-Level Fields

After specifying request information, add information about the individual items being purchased with the offer-level fields. The offer information is contained in a single field that contains pairs of field names and values. Name-value pairs are separated by carets (^), and names and values are separated by colons (:). Therefore, the values cannot contain carets and colons. You can include embedded spaces in the values, but if you include multiple consecutive spaces, only one of the spaces is retained when the request is processed.

You can include multiple offers in one request.

```
ics_fadd(icsorder, "offer0", "quantity:1^amount:45.95");
ics_fadd(icsorder, "offer1", "quantity:3^amount:14.99");
ics_fadd(icsorder, "offer2", "quantity:1^amount:25.95");
```

For a full list of the offer fields for the CyberSource credit card services, see *Credit Card Services Using the SCMP API* (PDF | HTML).

Sending the Request

You next send the request to CyberSource:

```
res = ics_send(icsorder);
```

Handling the Reply Flags and Error Messages

After the CyberSource server processes your request, the server returns a reply consisting of name-value pairs. The fields vary, depending on which services you requested and the results of the request.

To use the reply information, you must integrate it into your system and any other system that uses that data. This includes storing the data and passing it to any other services that need the information.

You must write an error handler to handle the reply flags and error messages that you receive from CyberSource. Do not show the flags or error messages directly to customers. Instead, present an appropriate response that tells customers the result.


Important

Because CyberSource may add reply fields and reason codes at any time, you should parse the reply data according to the names of the fields instead of their order in the reply.

The main reply fields to evaluate for the request are as follows:

- **ics_rcode** — a one-digit code indicating the result of the entire request:
  - **1** indicates the request is successful
  - **0** indicates the request is declined
  - **-1** indicates an error occurred
- **ics_rflag** — a one-word description of the result of the entire request:
  - **SOK** indicates the request was successful
  - A flag starting with the letter **D** if the request was declined, such as **DMISSINGFIELD**
  - A flag starting with the letter **E** if there was an error, such as **ESYSTEM**
- **ics_rmsg** — a message that explains the reply flag. Do not show this message to customers or use it to write the error handler.

You also receive similar fields for each service you request, indicating the result of the service. The names of the fields are **<*service*>_rcode**, **<*service*>_rflag**, and **<*service*>_rmsg**. For example, the service for credit card authorization (**ics_auth**) returns **auth_rcode**, **auth_rflag**, and **auth_rmsg**.


Important

CyberSource reserves the right to add new reply flags at any time. Write your error handler so that it can process these new reply flags without problems.

Receiving the Reply

Receive the reply (in the same statement used to send the request):

```
res = ics_send(icsorder);
```

Handling the Reply

Next, evaluate the request's reply code and reply flag:

```
if (res == NULL) {
  /* This happens only if ics_send cannot */
  /* allocate memory for the reply. */
  /* Log an error, notify customer of problem, and return. */
  return;
}
ics_print(res);
printf("-- end --\n");

rcode = ics_fgetbyname(res, "ics_rcode");
rflag = ics_fgetbyname(res, "ics_rflag");
rmsg = ics_fgetbyname(res, "ics_rmsg");
status = atoi(rcode);
switch (status) {
  case -1:
    /* Error occurred; Notify customer and ask to try again. */
    break;

  case 0:
    /* Request declined, evaluate rflag for reason why. */
    if (strcmp(rflag, "DINVALIDDATA") == 0) {
      /* Notify customer that request included invalid data. */
    }
    else if (strcmp(rflag, "DMISSINGFIELD") == 0) {
      /* Notify customer that request is missing a required field.*/
    }
    else if (strcmp(rflag, "DCARDREFUSED") == 0) {
      /* Card declined by the bank. Notify customer */
      /* unable to process order. */
    }
    else {
      /* Handle remaining declined reply flags here. */
      /* Write handler to handle new reply flags */
      /* created by CyberSource. */
    }
    break;
  case 1:
    /* Successful transaction; perform any local process */
    /* and notify customer of success. */
    break;
  default:
    /* Unexpected result; notify customer. */
}
```

Destroying the Request

Finally, destroy the used request and reply:

```
ics_destroy(icsorder);
ics_destroy(res);
```

Requesting Multiple Services

When you request multiple services in one request, CyberSource processes the services in a specific order. When a service fails, CyberSource does not process the subsequent services in the request.

For example, in the case of a sale (a credit card authorization and a capture requested together), if the authorization service fails, CyberSource does not process the capture service. The reply you receive only includes reply fields for the authorization.

Many CyberSource services include “ignore” fields that tell CyberSource to ignore the result from the first service when deciding whether to run the subsequent services. In the case of the sale, even though the issuing bank gives you an authorization code, CyberSource might decline the authorization based on the AVS or card verification results. Depending on your business needs, you might choose to capture these types of declined authorizations anyway. You can set the **ignore_avs** field to `"yes"` in your combined authorization and capture request:

```
ics_fadd(msg, "ignore_avs", "yes");
```

This tells CyberSource to continue processing the capture even if the AVS result causes CyberSource to decline the authorization. In this case you would then get reply fields for both the authorization and the capture in your reply.

Note

You are charged only for the services that CyberSource performs.

Timeouts and Automatic Retries

Retries and timeouts control how long the client waits for a reply from the CyberSource server after it sends a request, and whether and when the client automatically resends the request (called a retry). If you find that you are not receiving replies to some of your requests, you can troubleshoot the problem by adjusting how your code handles retries and timeouts.

Note

When the client retries, the retry request has the same request ID as the original request.

Three parameters allow you to control how retry requests and timeouts work:

- **retry_enabled** — indicates whether you want to send a retry request if you do not initially get a reply. Retry request is disabled by default.
- **retry_start** — controls how long (in seconds) you want to wait for the initial request to be sent and replied to by CyberSource. A retry request is sent if a reply is not received by the time this expires. The default value is `30`.
- **timeout** — controls the total amount of time (in seconds) you want to wait for a response before a timeout error is returned (whether retry was attempted or not). The default value is `110`.

Important

Retry is not attempted if the difference between the **timeout** and **retry_start** values is less than 3 seconds.

How a Transaction without Retry Works

When retry is disabled, the SDK proceeds as follows:

1. Encrypts and sends the initial request. The **timeout** count starts at 0 seconds.
2. Receives a response from the CyberSource server in the allotted by the **timeout** parameter.

   If the response does not come back from the CyberSource server in the allotted time, a timeout error is returned.

How a Transaction with Retry Works

When retry is enabled, the SDK proceeds as follows:

1. Encrypts and sends the initial request. The **timeout** and the **retry_start** counts start at 0 seconds.
2. Waits until **retry_start** has expired or returns the response if one was received, whichever comes first.
3. If **retry_start** has expired and no response has been received, the client encrypts and sends the request again, using the *same request ID* as the initial request. The **timeout** clock continues counting. The request and the response automatically include an additional field to indicate that this request is a retry.
4. The client returns a response if one is received, or waits until the full **timeout** has expired. (The **timeout** clock started counting at the time that the initial request was sent.)
5. If no response was received before the **timeout** expired, a timeout error is returned.

   For example, if **retry_start** is set at 30 seconds, and no response has been received in that 30 seconds, then the client sends a retry request.

   The client then waits for the duration of the remaining timeout time (**timeout** minus **retry_start**). For example, if **timeout** is set at 110 seconds, the client then waits for a length of time equal to 110 seconds minus 30 seconds. If no response is received in that 80-second interval, then a timeout error is returned.

Evaluating the Retry Reply Fields

You evaluate the success of the retry request using the **ics_retry** field. The field returns one of the following values:

- **1** — The retry request was successful using the original data; no reprocessing was necessary.

  The CyberSource server has a record of the response in its database. CyberSource does not reprocess the request and sends the original response to you.

- **0** — The retry request was successful using data from reprocessing the request.

  The CyberSource server has no record of the request. CyberSource reprocesses the request and sends you the response.

- **-1** — The retry request was unsuccessful due to a processing error.

  The CyberSource server has a record of the request, but the response is not in its database. CyberSource does not reprocess the request. Do not resend the request, because you might duplicate the transaction. Use Transaction Search in the Business Center to search for the transaction.

Setting Retry and Timeout Parameters

You can set the retry field values using the `ics_fadd()` function.

Enabling Retry

The **retry_enabled** value controls whether the client sends a retry request when no initial response is received. Retry is disabled by default. Enable **retry_enabled** with the following code:

```
ics_fadd(msg, "retry_enabled", "yes");
```

Setting the retry_start Value

The **retry_start** value controls how long the client waits before sending a retry request. The default value is 30 seconds. The minimum value is 3 seconds, which is acceptable for testing, but not for running live transactions. Start with the default value of 30 seconds and lower the value to 15 seconds if appropriate. The following code example sets the **retry_start** value to 15 seconds:

```
ics_fadd(msg, "retry_start", "15");
```

Setting the timeout Value

The **timeout** value controls the maximum time you want to wait to send the request and receive the response. This time begins when the client sends the initial request. The default value is 110 seconds and must not be less than the minimum value of 6 seconds when retry is enabled. The following code example sets the **timeout** value to 90 seconds

```
ics_fadd(msg, "timeout", "90");
```

Important

The **timeout** value must not be set at less than 6 seconds when retry is enabled. The **timeout** value must also be greater than the **retry_start** value by at least 3 seconds. You receive an error and the system sends no retry request when the **timeout** or **retry_start** values are too low or invalid.

System Errors and Retries

The client's automatic retry capability described above in "Timeouts and Automatic Retries," page 27 does not automatically retry in the case of system errors, only timeouts. You must design your transaction management system to include a way to correctly handle CyberSource system errors. System errors occur when you successfully receive a reply and the reply's **ics_rflag**=ESYSTEM. For more information about the ics_rflag, see "Handling the Reply Flags and Error Messages," page 23. Depending on which payment processor is handling the transaction, the ESYSTEM error may indicate a valid CyberSource system error or a processor rejection due to invalid data. In either case, CyberSource recommends that you do not design your system to retry sending a transaction many times when an ESYSTEM error occurs.

Instead, CyberSource recommends that you retry sending the request only two or three times with successively longer periods of time between each retry. For example, after the first system error response, wait 30 seconds and then retry sending the request. If you receive the same error a second time, wait one minute before you send the request again. Depending on the situation, you may decide you can retry sending the request after a longer time period. Determine what is most appropriate for your business situation.

If after several retry attempts you are still receiving a system error, it is possible that the error is actually being caused by a processor rejection and not a CyberSource system error. In that case, CyberSource recommends that you either:

- Search for the transaction in the Business Center, look at the description of the error on the Transaction Detail page, and call your processor to determine why they are rejecting the transaction.
- Contact CyberSource Customer Support to confirm whether your error is truly caused by a CyberSource system issue.

If TSYS Acquiring Solutions is your processor, you may want to follow the first suggestion because there are several common TSYS Acquiring Solutions processor responses that are returned to you as system errors and that only TSYS Acquiring Solutions can address.

C Functions

The CyberSource C library consists of several functions and one data type that you use to send and receive requests and replies.

ics_destroy() Function

This function clears a used `ics_msg`.

Table 2 ics_destroy() Function

Syntax	`void ics_destroy(ics_msg* msg)`
Description	Frees resources used by the specified `ics_msg` that was created by `ics_init()` or by `ics_send()`. Destroy used `ics_msgs` when your program no longer needs them.
Returns	Nothing.

ics_fadd() Function

Adds a field (a name-value pair) to a request.

Table 3 ics_fadd() Function

Syntax	`int ics_fadd(ics_msg* msg, char* name, char* value)`
Description	Adds the field name and its value to the specified request. Adding a field that is already in the request overwrites that field. You can also use `ics_fremove()` to remove a field before you add it again.
Returns	■ The number of fields in the request after the field is added. ■ `-1`: The field cannot be added because the number of added fields exceeds the maximum allowed number.

Example Adding a Field to a Request

```
qty = ics_fadd(request, "customer_firstname", "John");
```

ics_fcount() Function

Returns the quantity of fields in an `ics_msg`.

Table 4 ics_fcount() Function

Syntax	`int ics_fcount(ics_msg* msg)`
Description	Counts the number of fields in the specified request. Then use `ics_fget()` and `ics_fname()` to iterate through all the fields in the request.
Returns	The number of fields in the specified request.

ics_fget() Function

Returns the value of the name-value pair at the position in an `ics_msg` specified by index.

Table 5 ics_fget() Function

Syntax	`char* ics_fget(ics_msg* msg, int i)`
Description	Extracts the field value stored in the request at the position indicated by index. Use this to get all field values in an `ics_msg` by iterating from 0 to `ics_fcount(msg)-1`.
Returns	Returns a pointer to the value portion of the name-value pair at the position in the request specified by index.

Example Getting Field Names in a Request

```
int i, c;
char *name, *value;
c = ics_fcount(msg);
for (i = 0; i < c; i++) {
  name = ics_fname(msg, i);
  value = ics_fget(msg, i);
  printf("%s=%s\n", name, value);
}
```

ics_fgetbyname() Function

Returns a field from an `ics_msg` by name.

Table 6 ics_fgetbyname() Function

Syntax	*`char* ics_fgetbyname(ics_msg* msg, char* name)`*
Description	Extracts the field value from an *`ics_msg`* by its field name.
Returns	Pointer to the value portion of the name-value pair of the field with the specified name. Null: no field name exists in the specified request.

Example Accessing a Value in a Request

```
value = ics_fgetbyname(msg, "ics_rcode")
```

ics_fname() Function

Returns the name of the name-value pair at the position in an `ics_msg` specified by index.

Table 7 ics_fname() Function

Syntax	`char *ics_fname(ics_msg* msg, int i)`
Description	Extracts the field name stored in an `ics_msg` at the position indicated by index. Use this to get all field names in an `ics_msg` by iterating from the range of `0` to `ics_fcount(msg)-1`.
Returns	Pointer to the name portion of the name-value pair at the position in the request specified by index. `Null`: the index is out of the range.

Example Getting Field Names in a Request

```
int i, c;
char *name, *value;
c = ics_fcount(msg);
for (i = 0; i < c; i++) {
  name = ics_fname(msg, i);
  value = ics_fgetbyname(msg, name);
  printf("%s=%s\n", name, value);
}
```

ics_fremove() Function

Removes a field from a request.

Table 8 ics_fremove() Function

Syntax	`int ics_fremove(ics_msg* msg, char* name)`
Description	Removes the field with the specified name from the specified request. Use this call to change the value of a field by removing the field, then adding it using `ics_fadd()`.
Returns	The number of fields in the request after the field is removed. `-1`: The field is not removed or does not exist in the specified request.

ics_init() Function

Creates and initializes a request.

Table 9 ics_init() Function

Syntax	`ics_msg* ics_init(int debug_flag)`
Description	Creates and initializes a new instance of a CyberSource request. The `debug_flag` parameter is required (example: `int debug_flag = 0`). This parameter can contain one of the following values: ■ `0`: Debugging information not displayed (default). ■ `1`: `(ICS_DEBUG)` Requests that debugging information be sent to `stdout` whenever `ics_send` is called with this request. You can use `1` or `ICS_DEBUG`. ■ `2`: `(ICS_TRACE)` Writes debugging information to a log file. New logging information is appended to the existing log file, but does not overwrite it. You can use `2` or `ICS_TRACE`. For Linux and Solaris, the trace log file is `$ICSPATH/tmp/icsapilog.txt`. For Windows, the trace log file is `\icsapilog.txt` on the current drive from which the program is run. Debugging information includes: ■ Variable values ■ Fields of the request being sent ■ Encrypted data being sent ■ Data about where the request was sent ■ Encrypted data received as a result ■ Fields of the decrypted result **Important** CyberSource recommends that you use logging only when troubleshooting problems. To comply with all Payment Card Industry (PCI) and Payment Application Data Security Standards (PADSS) regarding the storage of credit card and card verification number data, the logs that are generated contain only masked credit card and card verification number data (CVV, CVC2, CVV2, CID, CVN). Follow these guidelines: ■ Use debugging temporarily for diagnostic purposes only. ■ If possible, use debugging only with test credit card numbers. ■ Never store clear text card verification numbers. ■ Delete the log files as soon as you no longer need them. ■ Never send email to CyberSource containing personal and account information, such as customers' names, addresses, credit card or checking account numbers, and card verification numbers. For more information about PCI and PABP (Payment Application Best Practices) requirements, see the Visa Cardholder Information Security Program.

Table 9 ics_init() Function (Continued)

Returns	Returns a pointer to `ics_msg`, or `NULL` if the function cannot allocate memory for `ics_msg`.

Example Creating a Request

```
request = ics_init(0);
```

ics_send() Function

Sends an encrypted request to the CyberSource server.

Table 10 ics_send() Function

Syntax	`ics_msg* ics_send(ics_msg* msg)`
Description	Sends an encrypted request to the CyberSource server. Each request consists of one field specifying the services to be performed, and several data fields required by the requested services. The returned message includes status fields and calculated data fields.
Returns	Pointer to a message which contains data about the successful or failed request. Null: Memory for the reply message cannot be allocated.

Example Sending a Request

```
result = ics_send(msg);
```

ics_set_config_file() Function

Sets a configuration file for a request.

Table 11 ics_set_config_file() Function

Syntax	`void ics_set_config_file(ics_msg* request, char* filename)`
Description	Sets a configuration file for a request. If the function fails to read the configuration file, then the request does not use a configuration file.
Returns	Nothing.

The following table lists parameters that you can use to specify fields you use for all requests in the configuration file. For more information about the configuration file, see "Creating the Configuration File," page 19.

Table 12 Configuration File Parameters

Parameter	Description and Example
`debug`	Enables debug output. Overrides the `ics_init()` parameter. You can use the following values: ■ `0`: Debugging information not displayed (default). ■ `1`: `(ICS_DEBUG)` Requests that debugging information be sent to `stdout` whenever `ics_send` is called with this request. You can use `1` or `ICS_DEBUG`. ■ `2`: `(ICS_TRACE)` Writes debugging information to a log file. New logging information is appended to the existing log file, but does not overwrite it. You can use `2` or `ICS_TRACE`. **Example:** debug=0
`ics.host_id`	Local IP address to use for generating a request ID. This value must not be `127.0.0.1`. Corresponds to the **host_id** request field. **Example:** `ics.host_id=67.187.183.26`
`ics.http_proxy`	HTTP proxy URL for the CyberSource server. **Example:** `ics.http_proxy=http://my.proxy.host:3128/`
`ics.http_proxy_password`	HTTP proxy password for the CyberSource server. **Example:** `ics.http_proxy_password=`*`myPassword`*
`ics.http_proxy_username`	HTTP proxy user name for the CyberSource server. **Example:** `ics.http_proxy_username=`*`myUserName`*
`ics.merchant_id`	Default CyberSource merchant identifier. Corresponds to the **merchant_id** request field. **Example:** `ics.merchant_id=`*`myMerchantID`*
`ics.server_host`	Default CyberSource server name. Corresponds to the **server_host** request field. **Example:** `ics.server_host=`*`ics2test.ic3.com`*

Table 12 Configuration File Parameters (Continued)

Parameter	Description and Example
`ics.server_port`	Default CyberSource server port. Corresponds to the **server_port** request field. **Example:** `ics.server_port=`*`80`*

ics_msg Data Type

A request or reply.

CHAPTER 3

Java Client

Important

The SCMP API clients are supported on 32-bit operating systems only.

Basic Java Program Example

Create a Java program to access CyberSource services. The example below shows the basic code required to send a request for credit card authorization and process the reply.

```
import com.cybersource.ics.client.*;
import com.cybersource.ics.client.message.*;
import com.cybersource.ics.base.message.*;
import com.cybersource.ics.base.exception.*;
import java.io.*;
import java.util.Properties;

public class CCAuthExample
{
   public static void main(String[] args)
      throws IOException, ICSException
   {
      // load basic properties from ICSClient.props file
      Properties props = new Properties();

      String propsfile = "/home/icsuser/CyberSource/ics_n.n.n/
      properties/ICSClient.props";
      props.load(new FileInputStream(new File(propsfile)));

      // create a client for CyberSource to send requests
      ICSClient client = new ICSClient(props);

      // create a request to hold customer information
      ICSClientRequest request = new ICSClientRequest();
      request.setField("ics_applications", "ics_auth");
      request.setField("customer_cc_number", "411111111111111");
      request.setField("customer_cc_expmo", "12");
```

```
        request.setField("customer_cc_expyr", "05");// set more fields as
        necessary
        // (not all required fields shown here)
        // create an offer to hold product information
        ICSClientOffer offer = new ICSClientOffer();
        offer.setField("amount", "14.95");
        offer.setField("quantity", "1");
        request.addOffer(offer);
        // send the request to CyberSource and receive the reply
        ICSReply reply = client.send(request);

        // get and handle the reply code and reply flag
        int rcode = reply.getReplyCode();
        String rflag = reply.getField("ics_rflag");
        if (rcode < 0) {
        // Error occurred;
        // Notify customer that an error occurred,
        // and ask customer to try again.
        }
        else if (rcode == 0) {
        // Request declined
        // Evaluate reply flag for reason why request was declined
        if ("DINVALIDDATA".equals(rflag)) {
        // Notify customer that request included invalid data;
        // Ask to correct error and resubmit request.
        }
        else if ("DCARDREFUSED".equals(rflag)) {
        // For cards declined by the bank.
        // Notify customer unable to process order.
        }
        else {
        // Handle remaining possible declined reply flags here.
        // Write error handler to process
        // new reply flags created by CyberSource.
        }
    }
    else {
        // Successful transaction;
        // perform any local processing, and
        // notify customer of success.
        }
    }
}
```

Installing and Configuring the SDK

This section explains how to install, configure, and test the SDK for Java. If you are a new customer installing the SDK for the first time, skip the section about upgrading and go directly to "Downloading the SDK File," page 42.

Minimum System Requirements

- Java 2 SDK version 1.2 or later
- 128 MB RAM allocated to your Java Virtual Machine (JVM)
- 10 MB of disk space
- A utility to uncompress the distribution file

The SDK supports UTF-8 encoding.

Important

Failure to configure your client API host to a unique, public IP address causes inconsistent transaction results.

The client API request ID algorithm uses a combination of IP address and system time, along with other values. In some architectures this combination might not yield unique identifiers. If it is not possible to configure your machines with unique IP addresses, CyberSource provides a client configuration parameter that you can use to identify your host. For the CyberSource SCMP API SDK for Java, this configuration parameter is the **ics.hostID** entry in the `ICSClient.props` file.

Upgrading the SDK

If you are an existing customer upgrading a previous version of the SDK, you follow many of the same procedures as a new customer installing the SDK for the first time.

The main difference is that you need not generate your certificate and private key files. Instead, copy your existing certificate and key files from your old SDK to the `ics_n.n.n/keys/` directory in the new SDK.

Do not, however, copy your old `ICSClient.props` properties file from the old SDK. Instead, use the `ICSClient.props.template` file provided in the `ics_n.n.n/properties/` directory of the new SDK. This ensures that you use the latest version of the file. Change the file to include the new location of your key files, and make any other changes you want, based on your old file. For more information, see "Indicating the Path to Your Security Keys," page 43 and "Setting Properties in ICSClient.props," page 44.

If you use the `ICSClient(Properties)` constructor in your code, make sure that it uses the correct path to your new `ICSClient.props` file. For more information, see "ICSClient(Properties)," page 63.

You can delete the old SDK directory after the new SDK is successfully installed and tested.

Downloading the SDK File

To download the SDK file:

Step 1 Create a target directory for the SDK files.

Step 2 Download the latest SDK package file from the downloads area of the CyberSource Support Center.

Step 3 Unzip or untar the package to your target directory.

The target directory contains the directories and files listed below.

Directory or File	Description
`doc/`	Contains Javadoc for the SDK.
`keys/`	Contains your certificate and private keys and the ECert application which can be used to generate your keys. Transaction security keys can also be generated in the Business Center. For more information, see *Creating and Using Security Keys* (PDF | HTML)
`properties/`	Contains the properties file that stores basic client properties required for transactions.
`test/`	Contains test programs for different types of transactions. Includes an `EJB/` directory that contains test programs for Enterprise JavaBeans™ (EJB) transactions.
`CHANGES.txt`	Describes the changes in this and previous versions of the SDK.
`LICENSE.txt`	Contains the CyberSource software license agreement.
`README_JDK12.txt`	Contains SDK installation and testing instructions, as well as frequently asked questions.
`README_EJB.txt`	Contains instructions for implementing and testing EJB server support. Describes the test programs in the `test/EJB/` directory.
`README_TEST.txt`	Contains descriptions of the different test programs in the `test/` directory.
`ics.jar`	Contains the Java class files for the SDK.

Setting the Classpath

You must set the CLASSPATH environment variable to include the `ics.jar` file.

Unix csh Environment

```
setenv CLASSPATH ${CLASSPATH}:<Install_dir>/ics_n.n.n/ics.jar
```

Unix sh Environment

```
export CLASSPATH=${CLASSPATH}:<Install_dir>/ics_n.n.n/ics.jar
```

Windows

To set the classpath for Windows operating systems:

Step 1 Right-click the My Computer icon.

Step 2 Click **Properties > Environment**.

Step 3 Edit the value for CLASSPATH to include `ics.jar`.

Example `C:\CyberSource\ics_n.n.n\ics.jar;.`

Indicating the Path to Your Security Keys

Important You must generate two transaction security keys—one for the CyberSource production environment and one for the test environment. For information about generating and using security keys, see *Creating and Using Security Keys* (PDF | HTML).

Use one of the three following options to indicate the path to your key files:

- Set the `ics.keysPath` property in the `ICSClient.props` file, for example:

  ```
  ics.keysPath=/opt/ics_n.n.n/keys/
  ```

  If you are using multiple merchant IDs, this method is the most efficient. Store the key files for all of your merchant IDs in one directory, indicated by the `ics.keysPath` property. CyberSource can then locate the key files automatically based on the merchant ID you specify in the request. This way you do not have to specify the individual paths for the files in the request or the properties file.

  You can override the path specified with `ics.keysPath` by using either of the other two options.

- Specify the full path for each file using the `myCert, myPrivateKey,` and `serverCert` properties in the `ICSClient.props` file, for example:

  ```
  myCert=/opt/ics/keys/MyMerchant.crt
  myPrivateKey=/opt/ics/keys/MyMerchant.pvt
  serverCert=/opt/ics/keys/CyberSource_SJC_US.crt
  ```

- Specify the full path for each file in the request itself. This overrides both of the other options. Use this method if you use multiple merchant IDs and do not store the key files for all of your merchant IDs in the same directory. For more information about setting fields in requests, see "Adding Request-Level Fields," page 51.

Setting Properties in ICSClient.props

Before you can send transactions to CyberSource, you must configure a properties file called `ICSClient.props`, located in the `properties/` directory in the SDK.

See Table 13 for a list of the properties that you can set. Using the properties file to specify these values is optional. If you do not specify a value in the properties file, you must set the value for the corresponding field in the request. For example, you can either set the `retryEnabled` property in the properties file, or you can set the **retry_enabled** field in the request itself. See "Adding Request-Level Fields," page 51 for more information about setting fields in the request.

In the `ICSClient.props` file, use this syntax to set property values: `fieldName=value`

Example ***myPrivateKey***`=/home/opt/ics_n.n.n/keys/example.pvt`

If you do not want to use one of the properties in the file, delete it or comment it out.

Table 13 ICSClient.props Entries

Properties	Explanation
`debugLevel`	Determines whether ICSClient writes debugging information. Use one of these values: ■ `0`: Debugging off (default) ■ `1`: Debugging information recorded ■ 2: Trace information recorded (less information than with `debugLevel=1`). **Important** CyberSource recommends that you use logging only when troubleshooting problems. To comply with all Payment Card Industry (PCI) and Payment Application (PA) Data Security Standards regarding the storage of credit card and card verification number data, the logs that are generated contain only masked credit card and card verification number data (CVV, CVC2, CVV2, CID, CVN). Follow these guidelines: ■ Use debugging temporarily for diagnostic purposes only. ■ If possible, use debugging only with test credit card numbers. ■ Never store clear text card verification numbers. ■ Delete the log files as soon as you no longer need them. ■ Never send email to CyberSource containing personal and account information, such as customers' names, addresses, credit card or checking account numbers, and card verification numbers. For more information about PCI and PABP requirements, see www.visa.com/cisp.
`debugFile`	Name of the file that stores debugging information when `debugLevel` is enabled. The default name is `debug.log`.
`ics.hostID`	IP address to use for generating a request ID. If none is specified, the computer's local IP address is used. Corresponds to the **host_id** request field.
`ics.keysPath`	Default location of the certificate and private key files if they are not specified as `myCert`, `myPrivateKey`, `serverCert`, or in the request. Corresponds to the **keys_path** request field.
`merchantID`	CyberSource merchant ID. Corresponds to the **merchant_id** request field.
`myCert`	Full path to your certificate file. Read only if `myCertData` property is not specified and certificate data is absent from the request message, for example: `/opt/java_sdk_ics_n.n.n/ics_n.n.n/keys/MyMerchant.crt` Corresponds to the **merchant_cert_file** request field.
`myCertData`	Default merchant certificate data if absent from the request message. The data must be Base-64 encoded. Corresponds to the **merchant_cert** request field.

Table 13 ICSClient.props Entries (Continued)

Properties	Explanation
`myPrivateKey`	Full path to your private key file. Read only if `myPrivateKey` property is not specified and private key data is absent from the request message, for example: `/opt/java_sdk_ics_n.n.n/ics_n.n.n/keys/MyMerchant.pvt` Corresponds to the **merchant_private_key_file** request field.
`myPrivateKeyData`	Default merchant certificate data if absent from the request message. The data must be Base-64 encoded. Corresponds to the **merchant_private_key** request field.
`serverCert`	Full path to the CyberSource server's certificate file, for example: `/opt/java_sdk_ics_n.n.n/ics_n.n.n/keys/CyberSource_SJC_US.crt` Corresponds to the **server_cert_file** request field.
`serverCertData`	Default server certificate data to use if absent from the request message. The data must be Base-64 encoded. Corresponds to the **server_cert** request field.
`serverName`	CyberSource server ID. The default value is `CyberSource_SJC_US`.
`serverURL`	Default URL for requests to CyberSource: ▪ Test: `http://ics2testa.ic3.com:80/` ▪ Production: `http://ics2a.ic3.com:80/` Default value: `http://ics2testa.ic3.com:80/` Corresponds to the **server_host** and **server_port** request fields.
`retryEnabled`	Set to `true` or `false`. The default value is `false`. For more information, see "Timeouts and Automatic Retries," page 58. Corresponds to the **retry_enabled** request field (which is set to `yes` or `no`).
`retryStart`	Number of seconds to wait before sending the retry request. The default value is `30`. For more information, see "Timeouts and Automatic Retries," page 58. Corresponds to the **retry_start** request field.
`timeout`	Client timeout value in seconds. The default value is `110`. For more information, see "Timeouts and Automatic Retries," page 58. Corresponds to the **timeout** request field.

Testing the SDK Installation

To test the SDK installation:

Step 1 Open a command prompt.

Step 2 In the directory where you extracted the SDK, go to the `test/` directory.

Step 3 To run the test application, type

```
java ICSClientTest
```

You receive the message `Transaction succeeded` if you successfully installed and configured the SDK. If you do not receive this message, verify that your certificate and private key files where generated correctly and that the location is specified correctly. For more information, see "Indicating the Path to Your Security Keys," page 43.

The SDK includes additional programs in the `test/` directory that you can use to test different CyberSource services. The `README_TEST.txt` file in the SDK describes the different programs.

Note When you are still in test mode, send your requests to the test server ics2test.ic3.com and use port 80. See the sample CyberSource requests for examples.

You should also intentionally fail test transactions. To do this, modify field values in the test programs in the `test/` directory of the SDK. You receive a message that says `Transaction Failed` and the reason for the error. The following examples show how to intentionally fail a test transaction.

Failure Due to Missing Field

To intentionally fail a test transaction due to a missing field:

Step 1 In a text editor, open the `ICSClientTest.java` file in the `test/` directory of the SDK.

Step 2 Change the statement that sets the billing city field so that the value is empty:

```
request.setBillCity("");
```

Step 3 Save and recompile the file.

Step 4 Run the test file:

a Open a command prompt.

b In the directory where you extracted the SDK, go to the `test/` directory.

c Type **`java ICSClientTest`**

You receive a message that says `Transaction Failed` and `DMISSINGFIELD`, indicating a missing field in the request.

Failure Due to Invalid Data

To intentionally fail a test transaction due to invalid data:

Step 1 In a text editor, open the `ICSAuthTest.java` file in the `test/` directory of the SDK.

Step 2 Change the statement that sets the customer credit card expiration year field so that the value is invalid, for example

```
request.setCustomerCreditCardExpirationYear("asdfjkl");
```

Step 3 Save and recompile the file.

Step 4 Run the test file:

a Open a command prompt.

b In the directory where you extracted the SDK, go to the `test/` directory.

c Type **`java ICSAuthTest`**

You receive a message that says `Transaction Failed` and `DINVALIDDATA`, indicating invalid data in one of the request fields.

Going Live

When you complete all of your system testing and are ready to accept real transactions from your customers, your deployment is ready to *go live*. When your deployment goes live, you can send transactions to CyberSource's production server. Provide your banking information to CyberSource so that your processor can deposit funds to your merchant bank account.

To send transactions to the CyberSource production server:

Step 1 Log in to the test Business Center.

Step 2 In the left navigation panel of the test Business Center, click **Support Center**.

Step 3 On the Support Center home page, in the **Search Knowledgebase** search well, type **go live**, and then click **Search**.

Step 4 In the search results, depending on what version of CyberSource you use, click **How do I go live? (Enterprise)** or **How do I go live? (Small Business)**.

These knowledgebase articles provide instructions that describe how to activate your account to process live transactions.

Once CyberSource has confirmed that your deployment is live, make sure to update your system so that you send requests to the production server (`ics2a.ic3.com`) instead of the test server (`ics2testa.ic3.com`).

After your deployment goes live, use real card numbers and other data to test every card type, currency, and CyberSource application that your integration supports. Because these are real transactions, use small monetary amounts to do the tests. Process an authorization, capture, and credit for each configuration. Use your bank statements to verify that money is deposited into and withdrawn from your merchant bank account. If you have more than one CyberSource merchant ID, test each one separately.

Using the SDK

This section explains how to request CyberSource services by using the Java SDK.

Requesting CyberSource Services

To request CyberSource services, you must:

- Create a system that collects the information required for the CyberSource services
- Create the ASP pages that do the following:
  - Assemble the order information into requests
  - Send the requests to the CyberSource server
  - Process the reply information

The CyberSource services use the SCMP API, which consists of name-value pair API fields. The name-value pair API fields you use for credit card orders are described in the *Credit Card Services Using the SCMP API* (PDF | HTML). The code in the examples in this section is incomplete. For complete sample programs, see the test programs in the `test/` directory of the SDK.

Constructing and Sending Requests

To access CyberSource services, you must create and send a request that holds the required information. You need instances of these methods:

- `ICSClient` — sends the request
- `ICSClientRequest` — holds the information about your company and the customer
- `ICSClientOffer` — holds the information about the item being purchased

This example shows basic code for requesting CyberSource services. In this example, Jane Smith is buying an item for $29.95.

Importing Classes

Add the following import statements:

```
import com.cybersource.ics.base.message.*;
import com.cybersource.ics.base.exception.*;
import com.cybersource.ics.client.message.*;
import com.cybersource.ics.client.*;
```

Creating the Client, Request, and Offer

You next create instances of the client, the request, and the offer, while handling a possible exception:

```
try {
ICSClient client = new ICSClient("../properties/ICSClient.props");
ICSClientRequest request = new ICSClientRequest();
ICSClientOffer offer = new ICSClientOffer();
} catch (ICSException e) {
System.out.println("Could not create client and/or message. ") +
    e.getMessage());
}
```

For more details about these classes and constructors, see "API for the Java SDK," page 62. You can use a single instance of a client to send multiple requests that use different merchant IDs. For more information, see "Using Multiple Merchant IDs," page 62.

Adding Services to the Request

You next indicate the service that you want to use by using the `setField(String fieldname, String value)` method:

```
request.setField("ics_applications", "ics_auth");
```

You can request multiple services by using commas to separate the service names. For example, you can request both credit card authorization and capture together (referred to as a “sale”) by using the following code:

```
request.setField("ics_applications", "ics_auth,ics_bill");
```

Adding Request-Level Fields

You next add request-level fields using the `setField(String fieldname, String value)` method. This includes basic information about the customer and your company. Some of the fields that you set in a request correspond to properties in the `ICSClient.props` file. Setting the values in the request overrides the values in the properties file.

If you request multiple services that share common fields, you must add the field only once.

```
request.setField("customer_firstname", "Jane");
request.setField("customer_lastname", "Smith");
request.setField("customer_cc_number", "4111111111111111");
```

The example above shows only a partial list of the fields required for the request. For a full list of the request fields for the CyberSource credit card services, see the *Credit Card Services Using the SCMP API* (PDF | HTML).

Adding an Offer to the Request

After you specify the request information, add the offer information about the individual items being purchased. You can include multiple offers in one request.

```
offer.setField("amount", "29.95");
request.addOffer(offer);
```

For a full list of the offer fields for the CyberSource credit card services, see the *Credit Card Services Using the SCMP API* (PDF | HTML).

Sending the Request

You next send the request to CyberSource while handling a possible exception:

```
try {
ICSReply reply = client.send(request);
} catch (ICSException e) {
System.out.println("Error while sending request: ") +
    e.getMessage());
}
```

For more details about the `ICSReply` class and the `send(ICSClientRequest)` method, see "API for the Java SDK," page 62.

Handling the Reply Flags and Error Messages

After the CyberSource server processes your request, the server returns a reply consisting of name-value pairs. The fields vary, depending on which services you requested and the results of the request.

To use the reply information, you must integrate it into your system and any other system that uses that data. This includes storing the data and passing it to any other services that need the information.

You must write an error handler to handle the reply flags and error messages that you receive from CyberSource. Do not show the flags or error messages directly to customers. Instead, present an appropriate response that tells customers the result.

Important Because CyberSource may add reply fields and reason codes at any time, you should parse the reply data according to the names of the fields instead of their order in the reply.

Each CyberSource service has its own list of reply flags and error responses that you must handle. For the credit card services, see the *Credit Card Services Using the SCMP API* (PDF | HTML) for a list of the fields. The main reply fields to evaluate for the request are as follows:

- **ics_rcode** — a one-digit code that indicates whether the entire request was successful:
  - **1** indicates the request was successful
  - **0** indicates the request was declined
  - **-1** indicates an error occurred
- **ics_rflag** — a one-word description of the result of the entire request:
  - **SOK** indicates the request was successful
  - A flag starting with the letter **D** indicates the request was declined
  - A flag starting with the letter **E** indicates there was an error
- **ics_rmsg** — a message that explains the reply flag. Do not show this message to customers or use it to write the error handler.

You also receive similar fields for each service you request, indicating the result of the service. The names of the fields are **<service>_rcode**, **<service>_rflag**, and **<service>_rmsg**. For example, the service for credit card authorization (**ics_auth**) returns **auth_rcode**, **auth_rflag**, and **auth_rmsg**.

Important CyberSource reserves the right to add new reply flags at any time. Write your error handler so that it can process these new reply flags without problems.

Receiving the Reply

Receive the reply (in the same statement you use to send the request) and get the request's **ics_rcode** and **ics_rflag** values:

```
try {
ICSReply reply = client.send(request);
} catch (ICSException e) {
System.out.println("Error while sending request: ") +
    e.getMessage());
}
int rcode = reply.getReplyCode();
String rflag = reply.getField("ics_rflag");
```

Handling the Reply

Next evaluate the request's reply code and reply flag:

```
if (rcode < 0) {
  // Error occurred;
  // Notify customer that an error occurred,
  // and ask customer to try again.
}
else if (rcode == 0) {
  // Request declined
  // Evaluate reply flag for reason why request was declined
  if ("DINVALIDDATA".equals(rflag)) {
    // Notify customer that request included invalid data;
    // Ask to correct error and resubmit request.
  }
  else if ("DCARDREFUSED".equals(rflag)) {
    // For cards declined by the bank.
    // Notify customer unable to process order.
  }
  else {
    // Handle remaining possible declined reply flags here.
    // Write error handler to process
    // new reply flags created by CyberSource.
  }
}
else {
  // Successful transaction;
  // perform any local processing, and
  // notify customer of success.
}
```

Handling Exceptions

You must handle several exceptions when constructing and sending requests. For a full list of the exceptions, see the Javadoc in the `doc/` directory of the SDK. At a minimum, you should catch `ICSException`, the base class for all of the exceptions.

Creating the Client and Request

The constructors for `ICSClient` and `ICSClientRequest` throw `ICSException`, the base class for all CyberSource exceptions. The following example gives sample code for catching `ICSException`.

```
try {
ICSClient client = new ICSClient();
ICSClientRequest request = new ICSClientRequest();
} catch (ICSException e) {
System.out.println("Could not create client and/or request. ") +
    e.getMessage());
}
```

`ICSClient` constructors also throw `ICSConfigException`, a subclass of `ICSException`, which indicates that a problem occurred with the property values.

Sending the Request

The common exceptions thrown when you send a request indicate either a network or an SDK configuration problem. The exceptions fall into two categories: those thrown before the server receives the request, and those thrown after.

Exceptions Thrown Before the Server Receives the Request

These two exceptions indicate that the CyberSource server did not receive the request:

- `ICSConfigException` — indicates a problem occurs with any of the property values
- `RequestMessageException` — indicates an error occurs when creating or sending the request

In these cases, you can send the same request again without risk of duplicating the transaction.

Exceptions Thrown After the Server Receives the Request

The next two exceptions indicate that the CyberSource server received the request, but a subsequent error occurred:

- `ICSMessageParseException` — indicates an error occurs when parsing the reply data
- `ReplyMessageException` — indicates an error occurs when decrypting or reading the reply

In these cases, you should not resend the request to avoid duplicating the transaction. Use Transaction Search in the Business Center to search for the transaction.

Typically you only must handle the four exceptions listed above. The following examples provide sample code.

```
try {
// send the request and receive the reply
ICSReply reply = client.send(request);
} catch (ICSConfigException e) {
System.out.println("Problem with CyberSource client configuration: ") +
    e.getMessage());
}
catch (RequestMessageException e) {
System.out.println("Could not send request to server: ") +
    e.getMessage());
}
catch (ICSMessageParseException e) {
System.out.println("Reply received, but could not parse it: ") +
    e.getMessage());
}
catch (ReplyMessageException e) {
System.out.println("Request sent; problem with reply: ") +
    e.getMessage());
}
```

Three of the four exceptions have subclasses that provide greater detail to help troubleshoot where in the code the error occurs. See the Javadoc in the `doc/` directory of the SDK for more information.

Requesting Multiple Services

When you request multiple services in one request, CyberSource processes the services in a specific order. If a service fails, CyberSource does not process the subsequent services in the request.

For example, in the case of a sale (a credit card authorization and a capture requested together), if the authorization service fails, CyberSource does not process the capture service. The reply you receive only includes reply fields for the authorization.

Many CyberSource services include “ignore” fields that tell CyberSource to ignore the result from the first service when deciding whether to run the subsequent services. In the case of the sale, even though the issuing bank gives you an authorization code, CyberSource might decline the authorization based on the AVS or card verification results. Depending on your business needs, you might choose to capture these types of declined authorizations anyway. You can set the **ignore_avs** field to `“yes”` in your combined authorization and capture request:

```
request.setField(“ignore_avs”, “yes”);
```

This tells CyberSource to continue processing the capture even if the AVS result causes CyberSource to decline the authorization. In this case you would then get reply fields for both the authorization and the capture in your reply.

Note You are charged only for the services that CyberSource performs.

Specifying the Proxy Settings in the Request

You can specify the proxy settings in the request, overriding the settings in the system properties. You might do this if you use the Java SDK in an application server environment with many applications that have unique socket connection requirements.

You can set the proxy host and port in the request message using the following code:

```
request.setField("http_proxy_host", "host");
request.setField("http_proxy_port", "port");
```

If you include the proxy settings in the request, the SDK uses the proxy for the connection. If you do not include the proxy settings in the request, the SDK uses the settings specified with the system properties `http.proxyHost` and `http.proxyPort`.

Important If the value you specify in the request for either the host or port is invalid, the proxy settings are silently ignored, and the SDK attempts a direct connection. If the firewall blocks the attempt and requires a proxy, the direct connection fails with an exception. If, however, the firewall does not block the attempt, the direct connection succeeds.

Timeouts and Automatic Retries

If you find that you are not receiving replies to some of your requests, you can troubleshoot the problem by adjusting how your code handles retries and timeouts.

Retries and timeouts control how long the client waits for a reply from the CyberSource server after it sends a request, and whether and when the client automatically resends the request (called a retry).

Note

When the client retries, the retry request has the same request ID as the original request.

Three parameters allow you to control how retry requests and timeouts work:

- **retry_enabled** — indicates whether you want to send a retry request if you do not initially get a reply. Retry request is disabled by default.
- **retry_start** — controls how long (in seconds) you want to wait for the initial request to be sent and replied to by CyberSource. A retry request is sent if a reply is not received by the time this expires. The default value is `30`.
- **timeout** — controls the total amount of time (in seconds) you want to wait for a response before a timeout error is returned (whether retry was attempted or not). The default value is `110`.

Important

Retry is not attempted if the difference between the **timeout** and **retry_start** values is less than 3 seconds.

How a Transaction without Retry Works

When retry is disabled, the SDK for Java functions as follows:

1. Encrypt and send the initial request. The **timeout** clock starts counting at 0 seconds.
2. Wait for and receive a response from the CyberSource server in the allotted time (set by the **timeout** parameter).

   If the response does not come back from the CyberSource server in the allotted time, a timeout error is returned.

How a Transaction with Retry Works

When retry is enabled, the SDK for Java functions as follows:

1. Encrypt and send the initial request. The **timeout** clock and the **retry_start** clock both start counting at 0 seconds.
2. Wait until **retry_start** has expired or return the response if one was received, whichever comes first.
3. If **retry_start** has expired and no response has been received, the client encrypts and sends the request again, using the *same request ID* as the initial request. The **timeout** clock continues counting.

   The request (and the response, if one is received) automatically includes an additional field to indicate that this request is a retry.
4. The client returns a response if one is received, or waits until the full **timeout** has expired. (The **timeout** clock started counting at the time that the initial request was sent.)
5. If no response was received before the **timeout** expired, a timeout error is returned.

   For example, if **retry_start** is set at 30 seconds, and no response has been received in that 30 seconds, then the client sends a retry request.

   The client then waits for the duration of the remaining timeout time (**timeout** minus **retry_start**). For example, if **timeout** is set at 110 seconds, the client then waits for a length of time equal to 110 seconds minus 30 seconds. If no response is received in that 80-second interval, then a timeout error is returned.

Evaluating the Retry Reply Fields

You evaluate the success of the retry request using the **ics_retry** field. The field returns one of the following values:

- **1** — The retry request was successful using the original data; no reprocessing was necessary.

  The CyberSource server has a record of the response in its database. CyberSource does not reprocess the request and sends the original response to you.
- **0** — The retry request was successful using data from reprocessing the request.

  The CyberSource server has no record of the request. CyberSource reprocesses the request and sends you the response.
- **-1** — The retry request was unsuccessful due to a processing error.

  The CyberSource server has a record of the request, but the response is not in its database. CyberSource does not reprocess the request. Do not resend the request, because you might duplicate the transaction. Use Transaction Search in the Business Center to search for the transaction.

Setting Retry and Timeout Parameters

You can set the parameters' values using the `setField(String fieldname, String value)` method. You can also set the parameters directly in the `ICSClient.props` file.

Enabling Retry

The **retry_enabled** value controls whether the client sends a retry request if no initial response is received. Retry is disabled by default. You enable **retry_enabled** with the following code (`"yes"` is case-insensitive):

```
request.setField("retry_enabled", "yes");
```

You can also set the value in the `ICSClient.props` properties file:

```
retryEnabled=true
```

Setting the retry_start Value

The **retry_start** value controls how long the client waits before sending a retry request. The default value is 30 seconds. The minimum value is 3 seconds, which is acceptable for testing, but not for running live transactions. Start with the default value of 30 seconds and lower the value to 15 seconds if appropriate.

You can set the **retry_start** value with the following code:

```
request.setField("retry_start", "15");
```

Or, you can set the value in the `ICSClient.props` properties file:

```
retryStart=15
```

Setting the timeout Value

The **timeout** value controls the maximum time you want to wait to send the request and receive the response. This time begins when the client sends the initial request. The default value is 110 seconds and must not be less than the minimum value of 6 seconds when retry is enabled.

You can set the **timeout** value with the following code:

```
request.setField("timeout", "90");
```

Or, you can set the value in the `ICSClient.props` properties file:

```
timeout=90
```

Important

The **timeout** value must not be set at less than 6 seconds when retry is enabled. The **timeout** value must also be greater than the **retry_start** value by at least 3 seconds. You receive an error and the system sends no retry request if the **timeout** or **retry_start** values are too low or invalid.

System Errors and Retries

The client's automatic retry capability described above in "Timeouts and Automatic Retries," page 58 does not automatically retry in the case of system errors, only timeouts. You must design your transaction management system to include a way to correctly handle CyberSource system errors. System errors occur when you successfully receive a reply and the reply's **ics_rflag**=ESYSTEM. For more information about the **ics_rflag**, see "Handling the Reply Flags and Error Messages," page 52. Depending on which payment processor is handling the transaction, the ESYSTEM error may indicate a valid CyberSource system error or a processor rejection due to invalid data. In either case, CyberSource recommends that you do not design your system to retry sending a transaction many times when an ESYSTEM error occurs.

Instead, CyberSource recommends that you retry sending the request only two or three times with successively longer periods of time between each retry. For example, after the first system error response, wait 30 seconds and then retry sending the request. If you receive the same error a second time, wait one minute before you send the request again. Depending on the situation, you may decide you can retry sending the request after a longer time period. Determine what is most appropriate for your business situation.

If after several retry attempts you are still receiving a system error, it is possible that the error is actually being caused by a processor rejection and not a CyberSource system error. In that case, CyberSource recommends that you use one of these options:

- Search for the transaction in the Business Center, look at the description of the error on the Transaction Detail page, and call your processor to determine if and why they are rejecting the transaction.
- Contact CyberSource Customer Support to confirm whether your error is truly caused by a CyberSource system issue.

If TSYS Acquiring Solutions is your processor, you may want to follow the first suggestion because there are several common TSYS Acquiring Solutions processor responses that are returned to you as system errors and that only TSYS Acquiring Solutions can address.

Enterprise JavaBeans™ (EJB)

The SDK includes an Enterprise JavaBeans™ (EJB) component to build server applications that access CyberSource services.

See the `README_EJB.txt` file in the SDK for details on creating the CdkApp application, packaging and deploying the CdkJAR bean, and testing your EJB implementation.

API for the Java SDK

This section describes the classes and class members that you use to send and receive CyberSource requests and replies. For information about all the classes in the SDK, see the Javadoc found in the `doc` directory of the SDK.

ICSClient

Declaration

```
public class ICSClient
extends java.lang.Object
implements java.io.Serializable
```

Description

The `ICSClient` class sends the application request to CyberSource for processing.

Using Multiple Merchant IDs

You can use one instance of `ICSClient` to send multiple requests that use different merchant IDs (for example, you might have a different merchant ID for each currency you use).

In previous versions of the SDK, you had to destroy the `ICSClient` object and instantiate a new one to use a different merchant ID. Starting with version 3.7.1, you need only one `ICSClient` instance, and you specify the merchant ID in the request itself.

Adding Properties to the Request

Each request requires basic properties, such as your certificate and private key. You can add the properties to the request in different ways, depending on your preference. Each `ICSClient` constructor adds the properties to the request differently.

Starting with version 3.7.1 of the SDK, all properties in the `ICSClient.props` file are optional and act as default values for some of the request fields. If you do not include values for these fields in the properties file, then you must include values in the request itself, or accept the default values.

For more information, see "Setting Properties in ICSClient.props," page 44.

Constructors

ICSClient()

```
public ICSClient()
```

The default constructor creates an instance with no default properties. Use this constructor when you want to supply all necessary information in the request message itself.

Throws

`ICSConfigException` — when a problem occurs with any of the property values

ICSClient(String)

```
public ICSClient(String propsFilename)
```

This constructor loads the properties file from the specified properties filename.

Parameter

`propsFilename` — the properties filename

Throws

`ICSConfigException` — when a problem occurs with any of the property values

ICSClient(Properties)

```
public ICSClient(Properties properties)
```

This constructor takes a Properties object for initialization. The Properties object must contain the properties listed in `ICSClient.props`.

Parameter

`properties` — the properties object containing the properties from the `ICSClient.props` file

Throws

`ICSConfigException` — when a problem occurs with any of the property values

The following examples illustrate two ways to use this constructor:

Example loading properties from the `ICSClient.props` file

```
Properties props = new Properties();
props.load(new FileInputStream(newFile("/opt/java_sdk_ics_n.n.n/
    ics_n.n.n/properties/ICSClient.props")));
ICSClient client = new ICSClient(props);
```

Example setting properties in code

```
Properties props = new Properties();
props.setProperty("myPrivateKey", "/home/icsuser/keys/
    MyMerchant.pvt");
props.setProperty("myCert", "/home/icsuser/keys/MyMerchant.crt");
props.setProperty("serverCert", "home/icsuser/keys/
    CyberSource_SJC_US.crt");
props.setProperty("merchantID", "MyMerchant");
props.setProperty("serverName", "CyberSource_SJC_US");
ICSClient client = new ICSClient(props);
```

`ICSClient` includes two additional constructors that are not recommended for use because they prevent the use of other default parameters, such as **timeout** and **retry_enabled.**

Method

send(ICSClientRequest)

`public ICSReply` **send**`(ICSClientRequest request)`

Sends the request to the CyberSource server and returns the reply in the form of an `ICSReply` object.

Parameter

`request` — the request

Throws

`ICSConfigException` — when a problem occurs with any of the property values

`RequestMessageException` — when an error occurs when creating or sending the request

`ICSMessageParseException` — when an error occurs when parsing the request data

`ReplyMessageException` — when an error occurs when decrypting or reading the reply

For more details about these exceptions, see "Handling Exceptions," page 55.

Three of the four exceptions have subclasses that provide greater detail to help troubleshoot where in the code the error occurs. See the Javadoc in the `doc/` directory of the SDK for more information.

Example sending a request to the CyberSource server

```
ICSReply reply = client.send(request);
```

ICSClientRequest

Declaration

```
public class ICSClientRequest
extends ICSRequest (which extends ICSMessage)
implements java.io.Serializable
```

Description

The `ICSClientRequest` class stores all the information in a request.

Constructor

ICSClientRequest()

```
public ICSClientRequest()
```

Use the default constructor to create an `ICSClientRequest` object.

Throws

`ICSException` — the base class for all CyberSource exceptions

Methods

Important

The `ICSClientRequest` convenience methods have been deprecated. Use the `setField` method instead.

setField(String, String)

```
public void setField(String name, String value)
```

Use the `setField(String, String)` method (inherited from the `ICSRequest` class) to assign values to the fields in the request.

Parameters

- `name` — name of the request field
- `value` — value for the request field

Example setting fields in the request

```
request.setField("ics_applications", "ics_auth");
```

addOffer(ICSOffer)

```
public void addOffer(ICSOffer i)
```

Use the `addOffer(ICSOffer)` method (inherited from the `ICSRequest` class) to add an offer to a request message.

Parameter

`i` — the offer

Example adding an offer to the request

```
request.addOffer(offer);
```

ICSClientOffer

Declaration

```
public class ICSClientOffer
extends ICSOffer (which extends ICSMessage)
```

Description

The `ICSClientOffer` class stores all the information in an offer, such as the product name, quantity, and cost.

Constructor

ICSClientOffer()

```
public ICSClientOffer()
```

Use the default constructor to create an `ICSClientOffer` object.

Method

Important

The `ICSClientOffer` convenience methods have been deprecated. Use the `setField` method instead.

setField(String, String)

```
public void setField(String name, String value)
```

Use the `setField(String, String)` method (inherited from the `ICSMessage` class) to assign values to the fields in the offer.

Parameters

- `name` — name of the offer field
- `value` — value of the offer field

Example Setting Fields in an Offer

```
offer.setField("amount", "14.95");
```

ICSReply

Declaration

```
public class ICSReply
extends ICSMessage
implements java.lang.Cloneable
```

Description

The `ICSReply` class stores the reply information you receive after the request is sent to CyberSource.

Example **using an `ICSReply` object to receive a reply**

```
ICSReply reply = client.send(request);
```

Method

getField(String)

```
public String getField(String field)
```

This method (inherited from `ICSMessage`) returns the value of the specified field. Use it to evaluate the fields contained in the reply.

Parameter

`field` — the field name

Example **getting the value of the reply flag**

```
String rflag = reply.getField("ics_rflag");
```

CHAPTER 4

.NET Client

Important

The SCMP API clients are supported on 32-bit operating systems only.

Online Help

The CyberSource SDK for .NET online help provides descriptions of the classes and functions of the .NET Client. To view the online help, open Microsoft Visual Studio .NET, open your project, then place your computer's pointer over the client's classes and its members in the code. If the code is not visible, right-click on a web page, then click **View Code**.

Basic C# Program Example

The following example shows the basic code for sending a transaction for tax calculation and processing the reply. This example is in `ICSTest.cs` file, located in the `ICSTest` directory. For more information on implementing this test code, see "," page 76.

```
using System;
using System.IO;
using CyberSource;
namespace ICSTest
{
  public class ICSTest
  {
    public static void Main( string[] args )
     {
      try

       {
        // If no argument is specified, use
        // ICSTest.xml in the working directory.
        string configFile = (args.Length > 0)
```

```
      ? args[0]
      : "ICSTest.xml";

    // create and populate request
    ICSRequest request = new ICSRequest();
    request["ics_applications"] = "ics_tax";
    request["merchant_ref_number"] = "12345";
    request["customer_firstname"] = "John";
    request["customer_lastname"] = "Doe";
    request["customer_phone"] = "6509656000";
    request["customer_email"] = "nobody@cybersource.com";
    request["customer_cc_number"] = "4111111111111111";
    request["customer_cc_expmo"] = "12";
    request["customer_cc_expyr"] = "05";
    request["bill_address1"] = "1295 Charleston Road";
    request["bill_city"] = "Mountain View";
    request["bill_state"] = "CA";
    request["bill_zip"] = "94043";
    request["bill_country"] = "US";
    request["offer0"] = "amount:1.43";
    request["currency"] = "USD";

     // print out request name-value pairs
    Console.WriteLine( "REQUEST:" );
    foreach( NameValuePair nvp in request)
    {
      Console.WriteLine( nvp );
    }

    // create client and send request
    ICSClient client = new ICSClient( configFile );


    ICSReply reply = client.Send( request );


    // print out reply name-value pairs
    Console.WriteLine( Environment.NewLine + "REPLY:" );
    foreach( NameValuePair nvp in reply)
   {
      Console.WriteLine( nvp );
   }
  }
  catch (BugException e)
{
    Console.WriteLine( Environment.NewLine + e );
}

  catch (ConfigIOException e)
{
    Console.WriteLine( Environment.NewLine + e );
}
```

```
      finally

      {
        Console.Write( Environment.NewLine + "Press <Enter>...");
        Console.Read();
      }
    }
  }
}
      catch (CriticalTransactionException e)

     {
        Console.WriteLine( Environment.NewLine + e );
     }
      catch (NonCriticalTransactionException e)
     {
        Console.WriteLine( Environment.NewLine + e );
      }
```

Installing and Testing the Client

This section explains how to install and test the SDK for .NET.

Minimum System Requirements

- Microsoft .NET Framework Redistributable, installed on your system

  For the minimum requirements for .NET, see the .NET home page, at http://msdn.microsoft.com/netframework/.

- Microsoft ASP.NET.

  This application is required for the Sample Store.

- 20 MB of disk space

  Microsoft Visual Studio .NET is optional.

The client supports ISO-8859-1 encoding by default. However, you can configure the client to use UTF-8 instead. See "Encoding," page 99.

Important

Failure to configure your client API host to a unique, public IP address causes inconsistent transaction results.

The client API request ID algorithm uses a combination of IP address and system time, along with other values. In some architectures this combination might not yield unique identifiers. If it is not possible to configure your machines with unique IP addresses, CyberSource provides a client configuration parameter that you can use to identify your host. For the CyberSource SCMP API SDK for .NET, this configuration parameter is **HostID**, which is an ICSClient property.

Installing the Client

The CyberSource .NET Client Setup Wizard creates program folder shortcuts.

To install the client:

Step 1 Download the latest .NET Client package from the Support Center.

Step 2 Run the downloaded file.

The Welcome to the CyberSource .NET Client Setup Wizard window appears.

Step 3 Click **Next** to continue.

The License Agreement window appears.

Step 4 Click **I Agree**, then click **Next** to continue.

The CyberSource .NET Client Information window appears.

Step 5 Click **Next**.

The Select Installation Folder window appears.

a In the bottom right corner of this window, click **Everyone** if you want all users on that computer to be able to view the Start menu program shortcuts to the CyberSource Client. If not, then click **Just Me**.

b To change the installation folder, click **Browse**, and choose a folder. To use the default selection, proceed to the next step.

Step 6 Click **Next**.

The Confirm Installation window appears.

Step 7 Click **Next**.

The Generate Keys window appears.

Step 8 In the Generate Keys window, click **Cancel**.

Important

The Generate Keys window key generation functionality has been deprecated. Instead, you must generate two security keys—one for the CyberSource production environment and one for the test environment—in the Business Center. For more information about generating and using transaction security keys, see *Creating and Using Security Keys* (PDF | HTML).

After you generate your security keys in the Business Center, you must edit the `ICSTest.xml`. This file must be updated before you run ICSTest. For more information, see "Editing the ICSTest.xml File," page 76.

Step 9 Click **OK**.

The Installation Complete window appears.

Step 10 Click **Close**.

Downloading the package installs the directory and files listed below.

Directory or File	Description
`ICSTest\`	Project and source files for the `ICSTest` test application.
`CyberSource.dll`	.NET assembly.
`CyberSource.xml`	CyberSource API help used by Intellisense in Visual Studio .NET.
`CybsSecurity.dll`	Security file.
`ecert.exe`	ECert application that generates your certificate and private keys. Transaction security keys can also be generated in the Business Center. For more information, see *Creating and Using Security Keys* (PDF | HTML).
`ICSTest.exe`	Test application that runs a CyberSource transaction to verify that the client is installed properly.
`LICENSE.rtf`	License Agreement for CyberSource software.
`README.rtf`	Documentation and registration information.
`StoreSetup.msi`	CyberSource Sample Store installer.

You are ready to install the CyberSource Sample Store.

Installing the CyberSource Sample Store

The CyberSource Sample Store Setup Wizard installs the CyberSource Sample Store on your computer.

Important The CyberSource Sample Store is intended for testing and basic demonstration purposes only. Do not use the CyberSource Sample Store to build your web store.

To install the CyberSource Sample Store:

Step 1 From your Windows desktop, click **Start** > **Programs** > **CyberSource .Net Client** > **Install Sample Store**.

The Welcome to the CyberSource Sample Store Setup Wizard window appears.

Step 2 Click **Next**.

The Select Installation Address window appears. The default virtual directory is `CyberSourceStore`. The default port is `80`.

Step 3 Click **Next**.

The Confirm Installation window appears.

Step 4 Click **Next**.

The Installation Complete window appears.


Note

If the Generate Keys window appears, click **Cancel**. The Generate Keys window key generation functionality has been deprecated. Instead, you must generate two security keys—one for the CyberSource production environment and one for the test environment—in the Business Center. For more information about generating and using transaction security keys, see *Creating and Using Security Keys* (PDF | HTML).

Step 5 Click **Close**.

Customize your Visual Studio .NET Toolbox.


Important

After you generate your security keys in the Business Center, you must specify the directory location of your private keys and certificate in the `web.config` file. For more information, see "Editing the web.config File," page 77.

Customizing Your Visual Studio .NET Toolbox

This section explains how to add the ICSClient component to your Visual Studio .NET Toolbox. To be able to drag and drop the CyberSource .NET Client into your web pages, you must add the ICSClient component to your Visual Studio .NET Toolbox.


Note

Remove the older ICSClient if there is one.

To customize the Visual Studio .NET Toolbox:

Step 1 From your Windows desktop, click **Start** > **Programs** > **Microsoft Visual Studio .NET** > **Microsoft Visual Studio .NET**.

Microsoft Visual Studio .NET opens.

Step 2 Click **Toolbox**.

If the Toolbox tab is not visible, then click **Toolbox** from the View menu.

Step 3 Right-click in the Toolbox window and click **Add/Remove Items**.

The Customize Toolbox window appears.

Step 4 Click **.NET Framework Components**.

The .NET Framework Components tab appears.

Step 5 Under the Name column, click **ICSClient**.

Step 6 Click **OK**.

The ICSClient icon is visible on the Components tab of the Tool menu.

You are ready to test your installation.

Testing the .NET Client Installation

To test your .NET Client installation, run the ICSTest application and run test transactions by using the Sample Store.

Important

You must generate two transaction security keys—one for the CyberSource production environment and one for the test environment. For information about generating and using security keys, see *Creating and Using Security Keys* (PDF | HTML).

Running ICSTest

If you did not update the `ICSTest.xml` file when you generated keys during the .NET Client installation, then you must edit the `ICSTest.xml` file before you run ICSTest application.

Editing the ICSTest.xml File

To edit the ICSTest.xml file:

Step 1 Using a text editor, open the `ICSTest.xml` file. The file is located in ***<directory>***`\ICSTest`, where ***<directory>*** is the directory where the CyberSource Client for .NET is installed.

Step 2 Find `<MerchantID>your_merchant_id</MerchantID>`, and change the value to your Merchant ID.

Step 3 Find `<KeysDir>directory_containing_your_keys</KeysDir>`, and change the value to the location of your public and private keys.

Step 4 Save and close the `ICSText.xml` file.

Running ICSTest

To run the ICSTest:

Step 1 From your Windows desktop, click **Start** > **Programs** > **CyberSource .NET Client** > **Run ICSTest**.

A Command Prompt window opens.

Step 2 Run the `ICSTest.exe` application.

If your installation was successful, the request and reply appear in the Command Prompt window.

If your transaction was not successful, make sure your certificate and private key files were properly generated. For more information, see *Creating and Using Security Keys* (PDF | HTML).

If you continue to receive an error message, review your `ICSTest.xml` file. For more information see "Editing the ICSTest.xml File," page 76.

The ICSTest directory contains the project and source files, which can be loaded in Visual Studio .NET. You must add a reference to the CyberSource .NET Client to build the project. For more information on adding a reference, see "Using the .NET Client with Other .NET Applications," page 84.

Using the Sample Store

If you did not update the `web.config` file when you generated keys during the .NET Client installation, then you must edit the `web.config` file before you use the Sample Store.

Editing the web.config File

To edit the web.config file:

Step 1 Using a text editor, open the `web.config` file. The file is located in the `inetpub\wwwroot\`***`<VirtualDir>`*** directory, where ***`<VirtualDir>`*** is the one you selected during the Sample Store installation. The default is `CyberSourceStore`.

Step 2 Find the `<appsettings>` element. It is similar to the following:

```
<appsettings>
...
</appsettings>
```

Step 3 Find `<add key="icsClient.KeysDir" value= "directory_containing_your_keys" />`, and change the value to the location of your public and private keys.

Step 4 Find `<add key="icsClient.MerchantId" value= "your_merchant_id" />`, and change the value to your merchant ID.

Step 5 Save and close the `web.config` file.

Using the Sample Store

To use the Sample Store:

Step 1 Using a web browser, go to `http://localhost/<virtualdirectory>/`, where ***<virtualdirectory>*** is the one you selected during the Sample Store installation.

The Shopping Cart page appears showing a pre-filled order.

Step 2 Click **Checkout**.

The Enter Shopper Information page appears. The field entries may be modified.

Step 3 Click **Submit**.

The Order Summary page appears. Tax and the order total are computed.

Step 4 Click **Proceed with Checkout**.

The Enter Credit Card Information page appears. Be sure the Credit Card Number is a test value. The Card Verification Number can be a three- or four-digit value.

Step 5 Click **Submit**.

If the .NET Client is working properly, the Order Complete page appears.

If you receive an error message, review your web.config. For more information see "Editing the web.config File," page 77.

Step 6 If you click **Shop** again, you are returned to the Shopping Cart.

Sample Store Files

The following files are in the Sample Store directory located in the Solution Explorer—My Store.

- `Default` — the Shopping Cart page
- `DisplayError` — shows the error page and prints out the exception
- `DisplaySuccess` — shows that your order is completed successfully, and displays the order number

- `DisplaySummary` — displays the total amount after tax; calls the **ics_tax** service and displays the Shopping Cart with tax, shipping and handling, and total amount
- `GetCreditCardInfo` — where the credit card information is displayed; calls the **ics_score** and **ics_auth** service after the Submit button is clicked.
- `GetShoppingInfo` — where the billing address and the ship-to address information are entered by the customer
- `Global.asax` — contains the ICSClient data member, which is used in `DisplaySummer.aspx` and `GetCreditCardInfo.aspx`


Note

The Sample Store also includes the project and source files, which can be loaded in Visual Studio .NET. You must add a reference to the CyberSource .NET Client to build the project. For more information about adding a reference, see "Using the .NET Client with Other .NET Applications," page 84.

Going Live

When you complete all of your system testing and are ready to accept real transactions from your customers, your deployment is ready to *go live*. When your deployment goes live, you can send transactions to CyberSource's production server. Provide your banking information to CyberSource so that your processor can deposit funds to your merchant bank account.

To send transactions to the CyberSource production server:

Step 1 Log in to the test Business Center.

Step 2 In the left navigation panel of the test Business Center, click **Support Center**.

Step 3 On the Support Center home page, in the **Search Knowledgebase** search well, type **go live**, and then click **Search**.

Step 4 In the search results, depending on what version of CyberSource you use, click **How do I go live? (Enterprise)** or **How do I go live? (Small Business)**.

These knowledgebase articles provide instructions that describe how to activate your account to process live transactions.

Once CyberSource has confirmed that your deployment is live, make sure to update your system so that you send requests to the production server (`ics2a.ic3.com`) instead of the test server (`ics2testa.ic3.com`).

After your deployment goes live, use real card numbers and other data to test every card type, currency, and CyberSource application that your integration supports. Because

these are real transactions, use small monetary amounts to do the tests. Process an authorization, capture, and credit for each configuration. Use your bank statements to verify that money is deposited into and withdrawn from your merchant bank account. If you have more than one CyberSource merchant ID, test each one separately.

Uninstalling the .NET Client

To uninstall the CyberSource client for .NET and the CyberSource Sample Store:

Step 1 From your Windows desktop, click **Start** > **Settings** > **Control Panel** > **Add/Remove Programs**.

The Add/Remove Programs window appears.

Step 2 In the left navigation bar, click **Change or Remove Programs**.

Step 3 Click **CyberSource Sample Store**.

Step 4 Click **Remove**.

The CyberSource Sample Store is removed.

Step 5 Click **CyberSource .NET Client**, noting the version number.

Step 6 Click **Remove**.

The CyberSource .NET Client is removed.

Step 7 Click **Close**.

Using the Client

This section explains how to use the .NET Client with ASP.NET and other .NET applications, and how to request CyberSource services.

Using the .NET Client With ASP .NET

When you use the .NET Client with ASP .NET, you can use `global.asax` or web form pages.

Putting it on `global.asax` allows you to use the same instance for different pages. Putting it on individual web form pages allows you to have a different configuration for each page.

You can also add more than one instance of ICSClient on one page, and give each instance a different variable name. For example, you can drag two instances of ICSClient into a page and each one uses a different merchant ID.

To use the .NET client with ASP .NET:

Step 1 From New Project, use the ASP .NET web application to create a new store.

Step 2 From the Component section of the Toolbox, drag the ICSClient component either to `global.asax` or a web form page.

Step 3 Copy `CybsSecurity.dll` from the CyberSource .NET Client installation directory to the bin directory of your ASP.NET application.

Step 4 Set its properties.

See "Setting Properties in ICSClient," page 81 for information on setting properties for ICSClient. See "Setting Dynamic Properties," page 83 for information on setting the properties as dynamic properties.

Step 5 Add code to the appropriate pages to send a transaction and handle the replies.

The CyberSource services use the SCMP API, which consists of name-value pair API fields. The name-value pair API fields you use for credit card orders are described in the *Credit Card Services Using the SCMP API* (PDF | HTML).

Note

For an existing application that uses a previous version of the CyberSource .NET Client, you must remove the reference to the older version and add a reference to the new one. For more information on adding a reference, see "Using the .NET Client with Other .NET Applications," page 84.

Setting Properties in ICSClient

Before you send transactions to CyberSource, configure the ICSClient's properties. You can configure properties either programatically or in the Properties window. To change properties in the Properties windows, choose a property in the left column, then enter a value for the property in the right column. For `LogLevel`, the property values are provided in a drop-down menu.

The following table lists the configurable properties of ICSClient.

Property	Description
`ConnectionLimit`	Maximum number of allowed concurrent connections between the client and CyberSource's server. For more information on this field and alternate ways to set the connection limits, see "Setting the Connection Limit," page 89.
`HostID`	IP address to use for generating **request_id**. By default, the client uses the machine's local IP address.
`HTTPProxyPassword`	Password used to authenticate against the HTTP proxy server. Used only when `HTTPProxyURL` is not null.
`HTTPProxyURL`	URL of your HTTP proxy server.
`HTTPProxyUsername`	Username used to authenticate against the HTTP proxy server. Used only when `HTTPProxyURL` is not null.
`KeysDir`	Directory containing the private key and certificates.
`LogFile`	Full path of the log file. The directory must already exist. Used only when `LogLevel` is not `LOG_NONE`.
`LogLevel`	Amount of data logged inside `ICSClient.Send()`. Possible values: ■ `LOG_CRITICAL_EXCEPTIONS`: When a CriticalTransactionException is thrown by `ICSClient.Send()`, it is logged with the configuration properties and, if available, the request and reply fields. ■ `LOG_EXCEPTIONS`: When any application exception is thrown by `ICSClient.Send()`, it is logged with the configuration properties and, if available, the request and reply fields. ■ `LOG_NONE`: No logging occurs. ■ `LOG_TRANSACTIONS`: The request and reply fields, with the configuration properties and any exceptions, are always logged. ■ `LOG_TRACES`: Trace statements are logged, in addition to the configuration properties and, if available, the request and reply fields and any exceptions. **Important** CyberSource recommends that you use logging only when troubleshooting problems. To comply with all Payment Card Industry (PCI) and Payment Application (PA) Data Security Standards regarding the storage of credit card and card verification number data, the logs that are generated contain only masked credit card and card verification number data (CVV, CVC2, CVV2, CID, CVN). Follow these guidelines: ■ Use debugging temporarily for diagnostic purposes only. ■ If possible, use debugging only with test credit card numbers. ■ Never store clear text card verification numbers. ■ Delete the log files as soon as you no longer need them. ■ Never send email to CyberSource containing personal and account information, such as customers' names, addresses, credit card or checking account numbers, and card verification numbers. For more information about PCI and PABP requirements, see www.visa.com/cisp.

Property	Description
`LogMaxSize`	The maximum size, in MB units, of the log file. When this size is reached, the file is renamed with the current timestamp, and a new file is created. Used only when the value of `LogLevel` is not `LOG_NONE`. The default value is `0`, which means unlimited.
`MerchantID`	Merchant identifier, used to retrieve your public key from CyberSource, assigned to you by CyberSource.
`RetryEnabled`	Indicates whether you want to attempt a retry request if necessary. The default value is `no`.
`RetryStart`	Number of seconds the system waits before attempting a retry request if retry is enabled. The default value is `30`. The minimum value is `3`.
`ServerHost`	Server host. The CyberSource test server, which is also the default value, is `ics2testa.ic3.com`. The CyberSource production server is `ics2a.ic3.com`
`ServerID`	ServerID, used to identify the key for encrypting requests. The default value is `CyberSource_SJC_US`.
`ServerPort`	Server port. The default value is `80`.
`ThrowLogException`	If set to `yes`, a LogException is thrown when logging fails. Use only when trying to troubleshoot the logging functionality. The default value is `no`.
`Timeout`	Number of seconds the system waits before returning a timeout error. The default value is `110`. `Timeout` must be greater than `RetryStart` by at least 3 seconds. The minimum value is 3 if retry is disabled, or 6 if retry is enabled.

Setting Dynamic Properties

Settings the ICSClient's properties as dynamic properties allows you to change selected properties of your web store without rebuilding your web application to implement the changes. You only must change their value in your store's `web.config`.

To set the ICSClient properties as dynamic properties:

Step 1 Right-click the ICSClient instance in the Visual Designer, and click **Properties**.

The Property Editor appear in the Properties window.

Step 2 In the far left column of the Properties window, click **Dynamic Properties** to expand the available options.

The Dynamic Properties for ICSClient window appears.

Step 3 Choose the properties you want to set as dynamic properties.

Step 4 Click **OK**.

Using the .NET Client with Other .NET Applications

Other non-ASP .NET applications can be used for CyberSource services such as **ics_bill**.


Note

When you add a new version of `CyberSource.dll`, remove the reference to the old version, then add a reference to the new version.

To use the .NET client with other .NET applications:

Step 1 In the Solution Explorer—My Store, click **My Store**, and then click **Reference**.

If the Solution Explorer is not visible, then click **Solution Explorer** from the View menu.

The Add Reference window appears.

Step 2 On the .NET tab, click **CyberSource .NET Client**.

Step 3 Click **OK**.

Step 4 Copy `CybsSecurity.dll` from the CyberSource .NET Client installation directory to the `bin\Debug and bin\Release` directories of your project.

Step 5 Write code to send transactions and handle the replies.

The CyberSource services use the SCMP API, which consists of name-value pair API fields. The name-value pair API fields you use for credit card orders are described in the *Credit Card Services Using the SCMP API* (PDF | HTML).

Requesting CyberSource Services

To request CyberSource services, you must write your code as follows:

- To collect the information that is required for the CyberSource services you use. The CyberSource services use the SCMP API, which consists of name-value pair API fields. The name-value pair API fields you use for credit card orders are described in the *Credit Card Services Using the SCMP API* (PDF | HTML).
- To assemble the order information into requests, send the requests to the CyberSource server, and process the replies.

Constructing and Sending Requests

To access any CyberSource service, you must create and send a request that holds the required information.

Importing the CyberSource Namespace

Add the following import statement

```
using CyberSource;
```

Creating a Request

Create an instance of ICSrequest:

```
ICSRequest request = new ICSRequest();
```

Adding Services to the Request

Next, add the service that you want to use:

```
request["ics_applications"] = "ics_tax";
```

You can request multiple services by using commas to separate the service names. For example, you can request both credit card authorization and capture together (referred to as a “sale”) by using the following code:

```
request["ics_applications"] = "ics_auth,ics_bill";
```

Adding Request-Level Fields to a Message

Add fields to the request:

```
request["merchant_ref_number"] = "12345";
request["customer_firstname"] = "John";
request["customer_lastname"] = "Doe";
request["customer_phone"] = "6509656000";
request["customer_email"] = "jdoe@example.com";
```

The example above shows only a partial list of the fields required for the request.

Adding Offer-Level Fields to a Message

Add the information about the individual items being purchased. An offer is a single field that contains pairs of field names and values:

```
request["offer0"] = "amount:1.43";
```

Or:

```
ICSOffer offer = new ICSOffer();
offer["amount"] = "1.43";
request.SetOffer(0,offer);
```

Examining Field Values

You can print the names and associated values of the fields you send to CyberSource:

```
// print out request name-value pairs
Console.WriteLine( "REQUEST:" );
foreach( NameValuePair nvp in request)
Console.WriteLine( nvp );
}
```

Sending the Request

You next send the request to CyberSource:

```
// create client and send request
ICSClient client = new ICSClient( configFile );
ICSReply reply = client.Send( request );
```

Examining Reply Fields

You can print the names and associated values of all the fields returned from CyberSource:

```
// print out reply name-value pairs
Console.WriteLine( Environment.NewLine + "REPLY:" );
foreach( NameValuePair nvp in reply)
{
  Console.WriteLine( nvp );
}
```

Handling Exceptions

You should handle the possible exceptions:

```
}
catch (BugException e)
{
  Console.WriteLine( Environment.NewLine + e );
}
catch (ConfigIOException e)
{
   Console.WriteLine( Environment.NewLine + e );
}
```

```
catch (CriticalTransactionException e)
{
  Console.WriteLine( Environment.NewLine + e );
}
catch (NonCriticalTransactionException e)
{
  Console.WriteLine( Environment.NewLine + e );
}
```

Handling Reply Flags and Error Messages

After the CyberSource server processes your request, the server returns a reply consisting of name-value pairs. The fields vary, depending on which services you requested and the results of the request.

To use the reply information, you must integrate it into your system and any other system that uses that data. This includes storing the data and passing it to any other services that need the information.

You must write an error handler to handle the reply flags and error messages that you receive from CyberSource. Do not show the flags or error messages directly to customers. Instead, present an appropriate response that tells customers the result.

Important Because CyberSource may add reply fields and reason codes at any time, you should parse the reply data according to the names of the fields instead of their order in the reply.

The main reply fields to evaluate for the request are as follows:

- **ics_rcode** — a one-digit code indicating the result of the entire request:
  - **1** indicates the request was successful
  - **0** indicates the request was declined
  - **-1** indicates an error occurred
- **ics_rflag** — a one-word description of the result of the entire request:
  - **SOK** indicates the request was successful
  - A flag starting with the letter **D** indicates the request was declined, such as **DMISSINGFIELD**
  - A flag starting with the letter **E** indicates there was an error, such as **ESYSTEM**
- **ics_rmsg** — a message that explains the reply flag. Do not show this message to customers or use it to write the error handler.

You also receive similar fields for each service you request, indicating the result of the service. The names of the fields are **<*service*>_rcode**, **<*service*>_rflag**, and **<*service*>_rmsg**. For example, the service for credit card authorization (**ics_auth**) returns **auth_rcode**, **auth_rflag**, and **auth_rmsg**.

Important CyberSource reserves the right to add new reply flags at any time. Write your error handler so that it can process these new reply flags without problems.

Each CyberSource service has its own list of reply flags and error responses that you must handle. For the credit card services, see the *Credit Card Services Using the SCMP API* (PDF | HTML) for a list of the fields.

Requesting Multiple Services

When you request multiple services in one request, CyberSource processes the services in a specific order. If a service fails, CyberSource does not process the subsequent services in the request.

For example, in the case of a sale (a credit card authorization and a capture requested together), if the authorization service fails, CyberSource does not process the capture service. The reply you receive only includes reply fields for the authorization.

Many CyberSource services include `ignore` fields that tell CyberSource to ignore the result from the first service when deciding whether to run the subsequent services. In the case of the sale, even though the issuing bank gives you an authorization code, CyberSource might decline the authorization based on the AVS or card verification results. Depending on your business needs, you might choose to capture these types of declined authorizations anyway. You can set the **ignore_avs** field to `"yes"` in your combined authorization and capture request:

```
request["ignore_avs"] = "yes";
```

This tells CyberSource to continue processing the capture even if the AVS result causes CyberSource to decline the authorization. In this case you would then get reply fields for both the authorization and the capture in your reply.

Note You are charged only for the services that CyberSource performs.

Setting the Connection Limit

This section explains how to increase the number of simultaneous connections between the client and CyberSource. By default, you can create only two simultaneous connections to an HTTP server. By increasing the number of connections, you can avoid a backlog of requests during times of very high transaction volume. Microsoft recommends for the connection limit a value that is 12 times the number of CPUs. For example, if you have two CPUs, you can set the connection limit to 24. Run performance tests to identify the optimum setting for your application.

Examples

You can increase the number of connections in many ways, for example by using an application- or server-specific configuration file where you can change the setting for a single or for all hosts. The examples below describe briefly some of the methods that you can use to increase connection limits.

cybs.connectionLimit

When set to a value other than `-1`, the `cybs.connectionLimit` setting in the client increases the limit for the host where you are sending the request by executing these statements on your behalf:

```
ServicePoint sp = ServicePointManager.FindServicePoint(uri);
sp.ConnectionLimit = config.ConnectionLimit;
```

<connectionManagement>

You can set the connection limit by using .NET's `<connectionManagement>` tag. In this example, the connection limit for CyberSource's test and production hosts is 12 while the limit for all other hosts is 2:

```
<system.net>
  <connectionManagement>
    <add address = "http://ics2testa.ic3.com" maxconnection = "12" />
    <add address = "http://ics2a.ic3.com" maxconnection = "12" />
    <add address = "*" maxconnection = "2" />
  </connectionManagement>
</system.net>
```

DefaultConnectionLimit

You can set the connection limit for all hosts to which your application is connected before a connection is made by using the following line in your start-up code:

```
ServicePointManager.DefaultConnectionLimit = your_value_here;
```

References

For more information on these and other methods to increase the connection limits, see the Microsoft documentation Managing Connections in the *.Net Framework Developer's Guide.*

Timeouts and Automatic Retries

If you find that you are not receiving replies to some of your requests, you can troubleshoot the problem by adjusting how your code handles retries and timeouts.

Retries and timeouts control how long the client waits for a reply from the CyberSource server after it sends a request, and whether and when the client automatically resends the request (called a retry).

Note

When the client retries, the retry request has the same request ID as the original request.

Three parameters allow you to control how retry requests and timeouts work:

- **retry_enabled** — indicates whether you want to send a retry request if you do not initially get a reply. Retry request is disabled by default.
- **retry_start** — controls how long (in seconds) you want to wait for the initial request to be sent and replied to by CyberSource. A retry request is sent if a reply is not received by the time this expires. The default value is `30`.
- **timeout** — controls the total amount of time (in seconds) you want to wait for a response before a timeout error is returned. The default value is `110`.

Important

Retry is not attempted if the difference between the **timeout** and **retry_start** values is less than 3 seconds.

How a Transaction without Automatic Retry Works

When retry is disabled, the SDK proceeds as follows:

1. Encrypts and sends the initial request. The **timeout** count starts at 0 seconds.
2. Receives a response from the CyberSource server in the allotted by the **timeout** parameter.

   If the response does not come back from the CyberSource server in the allotted time, a timeout error is returned.

How a Transaction with Automatic Retry Works

When retry is enabled, the SDK proceeds as follows:

1. Encrypts and sends the initial request. The **timeout** and the **retry_start** counts start at 0 seconds.
2. Waits until **retry_start** has expired or returns the response if one was received, whichever comes first.
3. If **retry_start** has expired and no response has been received, the client encrypts and sends the request again, using the *same request ID* as the initial request. The **timeout** clock continues counting.

   The request and the response automatically include an additional field to indicate that this request is a retry.
4. The client returns a response if one is received, or waits until the full **timeout** has expired. The **timeout** clock started counting at the time that the initial request was sent.
5. If no response was received before the **timeout** expired, a timeout error is returned.

   For example, if **retry_start** is set at 30 seconds, and no response has been received in that 30 seconds, then the client sends a retry request.

   The client then waits for the duration of the remaining timeout time (**timeout** minus **retry_start**). For example, if **timeout** is set at 110 seconds, the client then waits for a length of time equal to 110 seconds minus 30 seconds. If no response is received in that 80-second interval, then a timeout error is returned.

Evaluating the Retry Reply Fields

You evaluate the success of the retry request using the **ics_retry** field. The field returns one of the following values:

- **1** — The retry request was successful using the original data; no reprocessing was necessary.

  The CyberSource server has a record of the response in its database. CyberSource does not reprocess the request and sends the original response to you.

- **0** — The retry request was successful using data from reprocessing the request.

  The CyberSource server has no record of the request. CyberSource reprocesses the request and sends you the response.

- **-1** — The retry request was unsuccessful due to a processing error.

  The CyberSource server has a record of the request, but the response is not in its database. CyberSource does not reprocess the request. Do not resend the request, because you might duplicate the transaction. Use Transaction Search in the Business Center to search for the transaction.

Setting Automatic Retry and Timeout Parameters

Enabling Automatic Retry

The **retry_enabled** value controls whether the client sends a retry request if no initial response is received. Retry is disabled by default. You enable **retry_enabled** with the following code:

```
request["retry_enabled"] = "yes";
```

where "`yes`" is case-insensitive.

Setting the retry_start Value

The **retry_start** value controls how long the client waits before sending a retry request. The default value is 30 seconds. The minimum value is 3 seconds, which is acceptable for testing, but not for running live transactions. Start with the default value of 30 seconds and lower the value to 15 seconds if appropriate. You can set the **retry_start** value with the following code:

```
request["retry_start"] = "15";
```

Setting the timeout Value

The **timeout** value controls the maximum time you want to wait to send the request and receive the response. This time begins when the client sends the initial request. The default value is 110 seconds and must not be less than the minimum value of 6 seconds when retry is enabled. You can set the **timeout** value with the following code:

```
request["timeout"] = "90";
```

Important

The **timeout** value must not be set at less than 6 seconds when retry is enabled. The **timeout** value must also be greater than the **retry_start** value by at least 3 seconds. You receive an error and the system sends no retry request if the **timeout** or **retry_start** values are too low or invalid.

System Errors and Retries

The client's automatic retry capability described above in "Timeouts and Automatic Retries," page 91 does not automatically retry in the case of system errors, only timeouts. You must design your transaction management system to include a way to correctly handle CyberSource system errors. System errors occur when you successfully receive a reply and the reply's **ics_rflag**=ESYSTEM. For more information about the **ics_rflag**, see "Handling Reply Flags and Error Messages," page 87. Depending on which payment processor is handling the transaction, the ESYSTEM error may indicate a valid CyberSource system error or a processor rejection due to invalid data. In either case, CyberSource recommends that you do not design your system to retry sending a transaction many times when an ESYSTEM error occurs.

Instead, CyberSource recommends that you retry sending the request only two or three times with successively longer periods of time between each retry. For example, after the first system error response, wait 30 seconds and then retry sending the request. If you receive the same error a second time, wait one minute before you send the request again. Depending on the situation, you may decide you can retry sending the request after a longer time period. Determine what is most appropriate for your business situation.

If after several retry attempts you are still receiving a system error, it is possible that the error is actually being caused by a processor rejection and not a CyberSource system error. In that case, CyberSource recommends that you either:

- Search for the transaction in the Business Center, look at the description of the error on the Transaction Detail page, and call your processor to determine if and why they are rejecting the transaction.
- Contact CyberSource Customer Support to confirm whether your error is truly caused by a CyberSource system issue.

If TSYS Acquiring Solutions is your processor, you may want to follow the first suggestion because there are several common TSYS Acquiring Solutions processor responses that are returned to you as system errors and that only TSYS Acquiring Solutions can address.

API for the .NET Client

This section describes the C# classes, methods, and properties. Unless otherwise noted, all methods and properties are not thread safe. The exceptions thrown by some methods are followed by their exception codes, which are described in "Exceptions," page 102.

ICSClient

Description

The ICSClient class contains the methods that communicate to the CyberSource server. It is also a component that can be designed with Visual Studio .NET.

Constructors

ICSClient()

```
public ICSClient ()
```

This constructor sets all configuration properties to their default.

ICSClient(container)

```
public ICSClient ( System.ComponentModel.IContainer container )
```

This constructor initializes a new instance of the ICSClient class from the specified IContainer object. It is used by Visual Studio .NET when dragging the ICSClient component into a container such as a web forms page.

ICSClient(string)

```
public ICSClient ( string configFile )
```

This constructor reads the configuration properties from the specified file.

May Throw

```
BugException – INVALID_ARG, INVALID_CONFIG_FORMAT
ConfigIOException – LOAD
```

ICSClient(stream)

```
public ICSClient ( System.IO.Stream stream )
```

This constructor reads the configuration properties from the specified stream.

May Throw

```
BugException – INVALID_ARG, INVALID_CONFIG_FORMAT
ConfigIOException – LOAD
```

Properties

See "Setting Properties in ICSClient," page 81 for a list of the properties for ICSClient.

Methods

SaveConfig(string)

```
public void SaveConfig ( string configFile )
```

This method saves the configuration properties into the specified file.

May Throw

```
BugException – INVALID_ARG
ConfigIOException – SAVE
```

SaveConfig(stream)

```
public void SaveConfig ( Stream stream )
```

This method saves the configuration properties into the specified stream.

May Throw

```
BugException – INVALID_ARG
ConfigIOException – SAVE
```

ICSReply Send(request)

```
public ICSReply Send ( ICS Request request )
```

This method sends the request to the server and returns the reply. This method is safe to call concurrently as long as the ICSClient object's properties are not being modified from other threads at the same time.

May Throw

`BugException` — any possible `BugExceptionCode` value

`CriticalTransactionException` — any possible `CriticalTransactionExceptionCode` value

`LogException`

`NonCriticalTransactionException` — any possible `NonCriticalTransactionExceptionCode` value

ICSReply Send(request, logstream)

```
public ICSReply Send( ICS Request request, Stream logstream)
```

This method sends the request to the server and returns the reply. If the property LogLevel is not `LOG_NONE` and the logStream parameter is not `null`, all logging information is written to this stream object. If logStream is `null`, this method behaves the same as `ICSReply Send( ICSRequest request )`.

May Throw

`BugException` — any possible `BugExceptionCode` value

`CriticalTransactionException` — any possible `CriticalTransactionExceptionCode` value

`LogException`

`NonCriticalTransactionException` — any possible `NonCriticalTransactionExceptionCode` value

string ToString()

```
public override string ToString()
```

This method returns a string representation of the configuration properties.

ICSOffer

Description

The ICSOffer class encapsulates a CyberSource offer line. Instead of forming the offer line yourself, you can add the name-value pairs to an ICSOffer instance and then include the offer in the request by calling the SetOffer() method in the ICSRequest object.

Constructors

ICSOffer()

```
public ICSOffer()
```

This constructor initializes a new instance of the ICSOffer class.

Methods

Clear()

```
public void Clear()
```

This method erases all name-pair values previously set.

string Field(string)

```
public string Field( string field )
```

This method, which in C# is the indexer of the ICSOffer class, gets or sets the value of a field. If getting the value of the field, it is returned null if the field does not exist.

May Throw

`BugException – INVALID_ARG`

IEnumerator GetEnumerator()

```
public IEnumerator GetEnumerator()
```

This method returns an enumerator for the ICSOffer object.

string ToString()

```
public override string ToString()
```

This method returns a string representation of the ICSOffer object in the format accepted by the server.

ICSRequest

Description

The ICSRequest class encapsulates a request.

Constructors

ICSRequest()

```
public ICSRequest()
```

This constructor initializes a new instance of the ICSRequest class.

Property

Encoding

This property represents the System.Text.Encoding object used by ICSClient to convert the request into a stream of bytes. The default value is `Latin-1 (code page 850)`. To use UTF-8, set the encoding property as follows:

```
ICSRequest request = new ICSRequest();
request.Encoding = System.Text.UTF8Encoding;
```

Methods

Clear()

```
public void Clear()
```

This method erases all name-pair values previously set.

string Field(string)

```
public string Field( string field )
```

This method, which in C# is the indexer of the ICSRequest class, gets or sets the value of a field. If getting the value of the field, it is returned null if the field does not exist.

May Throw

`BugException – INVALID_ARG`

IEnumerator GetEnumerator()

```
public IEnumerator GetEnumerator()
```

This method returns an enumerator for the ICSOffer object.

SetOffer(int, offer)

```
public void SetOffer( int offerNo, ICSOffer offer )
```

This method adds or updates an offer line in the request via an ICSOffer object.

May Throw

`BugException – INVALID_ARG`

string ToString()

```
public override string ToString()
```

This method returns a string representation of the name-value pairs in the ICSRequest object.

ICSReply

Description

This class encapsulates the reply.

Methods

Clear()

```
public void Clear()
```

This method erases all name-pair values previously set.

string Field(string)

```
public string Field( string field )
```

This method, which in C# is the indexer of the ICSReply class, gets or sets the value of a field. If getting the value of the field, it returns null if the field does not exist.

May Throw

`BugException – INVALID_ARG`

IEnumerator GetEnumerator()

```
public IEnumerator GetEnumerator()
```

This method returns an enumerator for the ICSOffer object.

string ToString()

```
public override string ToString()
```

This method returns a string representation of the name-value pairs in the ICSRequest object.

NameValuePair Structure

Description

This structure represents each name-value pair when enumerating the name-value pairs inside the following classes:

- ICSRequest
- ICSOffer
- ICSReply

Fields

Name

The name of the field.

Value

The value of the field.

Methods

```
public override string ToString()
```

This method references a string representation of the name-value pair.

Exceptions

This section explains the exceptions thrown by some of the CyberSource .NET Client methods.

BugException

This exception is thrown when there is a problem in the code or in the configuration:

```
BugException has a property called ExceptionCode, which is of type
BugExceptionCode.
```

This table lists the values for `BugExceptionCode`.

BugExceptionCode Value	Description	Relevant Properties in the Exception Object
`INVALID_ARG`	Argument passed to a method is null or invalid.	`Message` — name of the argument.
`INVALID_CONFIG`	Property is null or invalid, preventing ICSClient from working properly.	`Message` — description of the problem.
`INVALID_CONFIG_FORMAT`	File or stream passed to ICSClient's constructor does not contain the correct format.	`InnerException` — actual exception that occurred.
`INVALID_FIELD`	Field in the request is null or invalid.	`Message` — field name and, if not null, its invalid value.
`DLL_NOT_FOUND`	CybsSecurity.dll cannot be found. Copy CybsSecurity.dll to the same location where the assembly CyberSource.dll is. For ASP.NET applications, this is usually the bin subdirectory. If the application is an executable, this is usually the same directory where the executable file is located.	`InnerException` — actual exception that occurred.
`INVALID_DLL`	Copy of CybsSecurity.dll is invalid. Contact CyberSource Customer Support for a valid copy.	`InnerException` — actual exception that occurred.

ConfigIOException

This exception is thrown when the loading or saving of the configuration data fails:

```
ConfigIOException has a property called ExceptionCode, which is of
type ConfigIOExceptionCode.
```

This table lists the values for `ConfigIOExceptionCode`.

ConfigIOExceptionCode Value	Description	Relevant Property in the Exception Object
LOAD	An IO error occurred while loading the configuration properties.	*`InnerException`* — actual exception that occurred.
SAVE	An IO error occurred while saving the configuration properties.	*`InnerException`* — actual exception that occurred.

CriticalTransactionException

This exception is thrown when an exception occurs after the CyberSource server has received and processed the request:

```
CriticalTransactionException has a property called ExceptionCode,
which is of type CriticalTransactionExceptionCode.
```

The transaction outcome is not known when this exception is thrown. Search for the transaction details in the Business Center to identify the transaction outcome.

This table lists the values for `CriticalTransactionExceptionCode`.

CriticalTransactionException-Code Value	Description	Relevant Property in the Exception Object
`HTTP_ERROR`	An HTTP response other than “200” was returned by the server.	`InnerException` — actual System.Net.WebException that occurred, whose Response property has the HttpWebResponse object.
`INCOMPLETE_REPLY`	The reply received from the server could not be parsed completely.	`Message` — remaining string in the reply that could not be parsed.
`PARSE_REPLY`	An error occurred while parsing the reply received from the server.	`Message` — whether it ran out of memory or parsing failed for some other reason, in which case it contains the raw reply received from the server.
`READ_RESPONSE_STREAM`	An error occurred while reading from the Response stream.	`InnerException` — actual System.IO.IOException that occurred.
`RECEIVE`	An error occurred while receiving the reply from the server.	`InnerException` — actual System.Net.WebException that occurred.

CriticalTransactionException-Code Value	Description	Relevant Property in the Exception Object
TIMEOUT	The request timed out.	`InnerException` — actual System.Net.WebException that occurred.
UNKNOWN_STATUS	An error occurred while transmitting the request and it is unknown whether the server received the request.	`InnerException` — actual System.Net.WebException that occurred.

LogException

This exception is thrown when the property `ThrowLogException` in the Client object is set to `yes`, and an exception occurred during logging.

The property `Message` contains the entire text that would have been logged if the exception did not occur. The property `InnerException` contains the actual exception that occurred.

If an exception was being logged when the LogException occurred, the property `ExceptionBeingLogged` (of type System.ApplicationException) contains this exception.

NonCriticalTransactionException

This exception is thrown when an exception occurs before the CyberSource server has received the entire request.

`NonCriticalTransactionException` has a property called `ExceptionCode`, which is of type `NonCriticalTransactionExceptionCode.`

This table lists the values for `NonCriticalTransactionExceptionCode`.

NonCriticalTransaction-ExceptionCode Value	Description	Relevant Properties in the Exception Object
COMPOSE_REQUEST	An error occurred while composing the request.	`Message` — whether it ran out of memory or the composition failed for some other reason.
CONNECT	Could not connect to the server.	`InnerException` — actual System.Net.WebException object.
GENERATE_REQUEST_ID	Could not generate a request ID.	`Message` — description of the problem. `InnerException` — if applicable, refers to a System.Net.Sockets.SocketException object.
GET_REQUEST_STREAM	An error occurred while obtaining the request stream.	`InnerException` — actual System.IO.IOException object.

NonCriticalTransaction-ExceptionCode Value	Description	Relevant Properties in the Exception Object
LOAD_KEY	An error occurred while loading a key.	`Message` — full path of the key that failed to load. `InnerException` — actual System.IO.IOException object.
PROXY_AUTH	Failed to authenticate against the proxy server.	`InnerException` — actual System.Net.WebException object.
PROXY_CONNECT	An error occurred while connecting to the proxy server.	`InnerException` — actual System.Net.WebException object.
SEND	An error occurred while sending the request to the server.	`InnerException` — actual System.Net.WebException object.
WRITE_REQUEST_STREAM	An error occurred while writing to the request stream.	`InnerException` — actual System.Net.WebException object.

Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Symbols

C

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

J