Developer's Handbook

Netscape LivePayment

Version 1.0


Recycled and Recyclable Paper

The Team:

Engineering: Choong Chew, Rich Coulter, Jane Ha, Terry Hayes, Jo-ning Ta, Nasrul Islam, Abhinendan Prateek, Roscoe Shih, Ed Andrews
Release Engineering: Patrick Marion
Marketing: Basil Hashem, Ammiel Kamon
Publications: Bill Branca, Ann Hillesland, Cindy Hall, Paul Ferlito, Meera Holla
Quality Assurance: Robert Bruce, Dan Findley, Peeter Pirn, Colin Wiel, Rick Yamaura
Technical Support: Fanny Wu, Dan Yang

Version 1.0

Printed in USA
97 96 95 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

About this book 7
Before you begin 7
Audience 7
Organization 8
Conventions 10

Part 1 LivePayment basics

Chapter 1 Introducing Netscape LivePayment 13
What is Netscape LivePayment? 14
Processing credit cards with Netscape LivePayment 16
Developing a payment processing application 18
Modifying the Starter Application Set 19
Creating your own application using LiveWire 19
Creating your own application using cpcmd 20
Certifying your application 20
Data storage 20
Applying for service from your bank 21
Security 21
Further reading on security 23

Chapter 2 Setting up Netscape LivePayment 25
Post-installation procedures 26
Setting up a relationship with a bank card acquirer 27
Changing LivePayment operating modes 28
Starting up in loopback mode 29
Changing from loopback mode to test mode 29
Changing from test mode to production mode 30

Accessing the Netscape LivePayment page 30
Configuring the LivePayment parameters 32
Configuring the card processor parameters 33
Domain Name Service (DNS) aliasing 35
Configuring security 35
Using the server's certificate 36
Using a separate certificate for LivePayment 38
Administering Netscape LivePayment 45
Starting the card processor 46
Checking the gateway connection 46
Stopping the card processor 47
Administering LiveWire 47
Accessing the Server Selector 49

Chapter 3 Payment application concepts 51
Credit card transactions 52
Creating a slip 52
Authorizing a purchase 53
Starting a new batch 53
Capturing a purchase 54
Issuing credit for a purchase 54
Settling a batch 55
Business rules for credit card applications 55
Authorize 55
Capture 56
Credit 56
Settle 57
Designing your database 57
Using batches 58
Using batches in an application 58
Order of transactions 60
Storing Information from the acquirer 64
Results of authorize 64
Batch number 65

Maintaining the payment and batch states 65
Why is the state needed? 66
Keeping track of the payment states 66
Keeping track of the batch states 67
Idempotent transactions 68

Part 2 Using the LivePayment Starter Application Set

Chapter 4 Running the Starter Application Set 73
The application files 74
Before you begin 75
Prerequisites 76
Configuration 76
Starting the application 76
Making a purchase 77
Using the administration interface 77
Viewing uncaptured authorizations 78
Viewing transactions in the current batch 78
Cancelling a transaction 78
Crediting a transaction by batch 79
Manually crediting a transaction 79
Viewing previous transactions by batch 80
Settling the current batch 80
Searching for a transaction 80

Chapter 5 Reviewing the application code 83
The credit card processing functions 84
The LiveWire objects 84
The database object 85
The project object 86
The request object 86
The LivePayment objects 86
The Merchant object 87
The Processor object 87
The Slip object 87

The Batch object ... 87
The Terminal object ... 88
Authorizing a purchase ... 88
Generating a slip ... 88
Encoding a slip ... 89
Authorizing a transaction ... 90
Capturing a transaction ... 91
Crediting a transaction ... 92
Settling a batch ... 92

Chapter 6 Modifying the Starter Application Set ... 95
Preparing to modify LPStart ... 96
Modifying the Starter Application Set ... 96
Modifying LPStart ... 96

Chapter 7 Netscape LivePayment Starter Application Set Reference ... 103
Directory Structure ... 104
LPStart HTML pages ... 105
LPAdmin HTML pages ... 105
LPAdmin Transaction Libraries ... 107
LPAdmin Merchant Libraries ... 108
Database schema ... 109
Database tables ... 110
Functions ... 114
Batch states ... 138
Transaction states ... 138

Part 3 Creating a LivePayment application from the ground up

Chapter 8 Using the LivePayment objects ... 143
LivePayment objects overview ... 144
Developing with LivePayment ... 144
Embedding LivePayment JavaScript in HTML ... 146
Registering LivePayment objects in LiveWire ... 146
Creating instances of LivePayment objects ... 148

Using LivePayment properties 148
Using the default values from the LivePayment configuration 149
Using error status methods 150
Getting the LivePayment version in LiveWire 152
Using LivePayment objects to process payments 152
Creating Merchant and Terminal objects 152
Creating a Processor object 154
Creating a Batch object 155
Creating a Slip object 157
The order description and the merchant order description 160
Encoding and decoding a Slip object 160
Creating a PayEvent object 162
Authorizing a payment 164
Capturing a payment 165
Crediting an account 166
Settling payments 166
Using the LPAuthOnly sample application 167
The LPAuthOnly files 167
Running LPAuthOnly 168
The sample code 170

Chapter 9 LivePayment object reference 173

Part 4 Creating an application using the cpcmd utility

Chapter 10 Using the cpcmd utility 209
Overview of cpcmd 210
Transaction flow 210
Default configuration 211
Data storage 212
Return values, output, and errors 212
Using TraceFile and TraceLevel 213
Command reference 214
Authorize 214
Capture 217

CreateSlip 219
Credit 221
GetCurrentBatch 223
SettleBatch 223

Part 5 Appendices

Appendix A Troubleshooting LivePayment 229
Troubleshooting overview 230
Resolving card processor error messages 230
Table notes 241
Verifying the configuration 242
Testing the gateway connection 243
Using traceroute and telnet 243
Using traceroute 244
Using telnet 245
Checking the card processor log file 245
Technical support 247

Appendix B Netscape-supported bank card acquirers 249
First Data Corporation (FDC) 249
Establishing a credit card business agreement 249
Obtaining test processing parameters 250
Getting information about your certificate 251
Certifying your application 251

Index 253

About this book

This *Handbook* describes the operations of Netscape LivePayment, Version 1.0, from Netscape Communications Corporation.

This Introduction discusses the intended audience, the organization, and provides a listing of typographic conventions used in this document. If you spend a few minutes looking through the Introduction before reading the rest of the *Handbook*, you will be able to utilize the *Handbook* more effectively.

Before you begin

This manual is written with the assumption that you understand the operating system on which you are running this software.

You do not need to be an expert on the Internet, the World Wide Web, or HTML, but you will find it helps to know the basics of these technologies.

This manual assumes that you have read the Netscape LiveWire documentation and the documentation for your Netscape server.

Audience

This manual is written for developers, merchants using LivePayment for commerce on the Internet, and for administrators of the commerce sites. The web site developers should have some programming experience with a programming language such as Pascal, C, or Visual Basic.

Organization

This section presents a brief summary each chapter making up this *Handbook*.

Part 1: LivePayment basics

Chapter 1: Introducing Netscape LivePayment

This chapter provides a brief introduction to the Netscape LivePayment system. It includes an overview of LivePayment concepts, and the basic architecture of the LivePayment payment system.

Chapter 2: Setting up Netscape LivePayment

This chapter contains information on setting up Netscape LivePayment after it has been installed. It contains information on setting up a relationship with a bank card acquirer, configuring security, and administering Netscape LivePayment.

Chapter 3: Payment application concepts

This chapter describes the transaction flow for processing credit cards and other information about developing with LivePayment

Part 2: Using the LivePayment Starter Application Set

Chapter 4: Running the Starter Application Set

This chapter describes running the Starter Application Set, from the viewpoint of a customer running the applications and a merchant administering the applications.

Chapter 5: Reviewing the application code

This chapter describes the Starter Application Set's code for credit card transactions.

Chapter 6: Modifying the Starter Application Set

This chapter describes modifying the Starter Application Set to create your own LivePayment application.

Chapter 7: Netscape LivePayment Starter Application Set Reference

This chapter contains reference information on the Starter Application Set's functions and database fields.

Part 3: Creating a LivePayment application from the ground up

Chapter 8: Using the LivePayment objects

This chapter provides an overview and description of the LivePayment objects.

Chapter 9: LivePayment object reference

This chapter contains reference information on each LivePayment object, property, and method.

Part 4: Creating an application using the cpcmd utility

Chapter 10: Using the cpcmd utility

This chapter contains information on the **cpcmd** utility.

Part 5: Appendices

Appendix A: Troubleshooting LivePayment

This chapter contains information on troubleshooting the connection between the card processor and the bank card acquirer.

Appendix B: Netscape-supported bank card acquirers

This chapter contains information on Netscape-supported bank card acquirers and how to contact them.

Conventions

A number of typographic conventions are used throughout this manual to help you recognize special terms and instructions. These conventions are summarized in the table below.

Convention	Meaning	Example
boldface	items on the screen	Click the **OK** button to configure the card processor.
	names of keys	Press **Enter** to clear the message.
	methods, functions, objects and properties	The example uses the **bad** method to test for an error.
boldface numbered steps	higher level descriptions of tasks you perform (more detailed instructions follow)	**1. Enter the group information.** Enter the name in the **Group Name** field, and a short description in the **Description** field.
italics	key words, such as terms that are defined in the text	If the transaction is authorized, a *capture* takes place.
	names of books	For more information, see the *LiveWire Developer's Guide.*
	variables (placeholders)	`terminalObject = new Terminal("terminalNumber")`
`courier font`	command line input or output	Change to the LivePayment configuration directory. `install_dir/admin/config`
	text file content, such as HTML templates and configuration files	`<HTML>` `<TITLE>Netscape LivePayment</TITLE>`
	code samples	`mer = new Merchant();` `term = new Terminal();`

1

LivePayment basics

- **Introducing Netscape LivePayment**
- **Setting up Netscape LivePayment**
- **Payment application concepts**

Chapter

1

Introducing Netscape LivePayment

This chapter provides a basic description and conceptual overview of Netscape LivePayment.

This chapter has the following sections:

- What is Netscape LivePayment?
- Processing credit cards with Netscape LivePayment
- Applying for service from your bank
- Security

What is Netscape LivePayment?

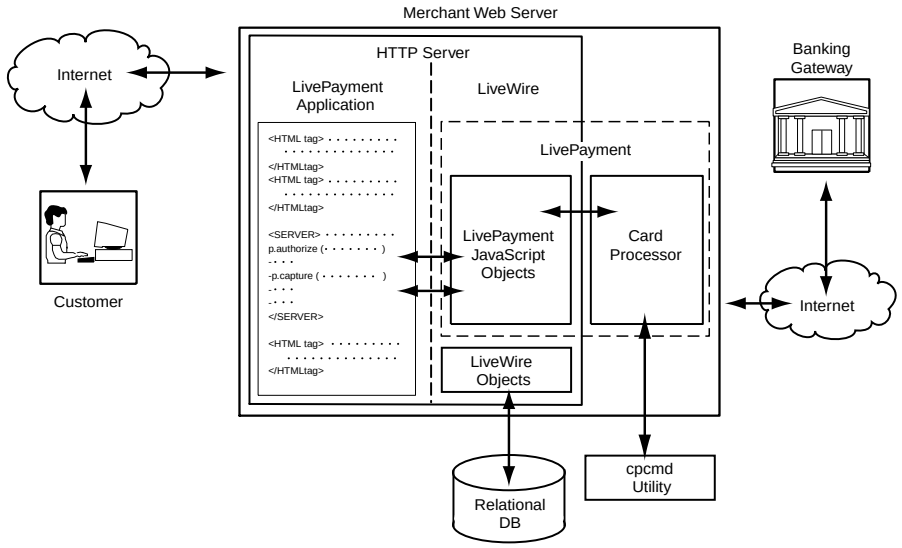
Netscape LivePayment provides a convenient way for you to create a Web site that executes financial transactions over the Internet. A system using LivePayment consists of:

- A Netscape Enterprise Server (Web server)
- Netscape LiveWire (a development environment)
- A relational database (an Informix database is included with LiveWire Pro)
- A card processor which routes financial transaction information to banks over the Internet and receives replies
- A Starter Application Set which you can modify to create your own payment application
- A set of LiveWire objects for credit card processing
- A utility for credit card processing

These building blocks make it simple to set up a Web site that accepts credit card payments. These components make it easy to customize your site using server-side JavaScript.

Figure 1.1 illustrates Netscape's payment system.

Figure 1.1 Netscape Payment System



The customer, using a browser, accesses an online form. The customer enters payment information into it. Using SSL security, the browser communicates the information to the *HTTP server*, the Netscape server that runs the Web site. From the server, the transaction information goes either to a LiveWire application, or to an interface that runs the **cpcmd** (card processor command) utility.

LiveWire is a Netscape development environment which includes authoring, scripting, and database access capabilities. The LiveWire scripting language is called *JavaScript*. JavaScript is an easy, Java-compatible scripting language provided with LiveWire. You can create a LiveWire application that processes payments either by modifying the Starter Application Set shipped with the product, or by creating your own application using the LivePayment objects that perform credit card transactions.

LivePayment also contains a *utility* (**cpcmd**), which has several commands that perform credit card transactions. The utility is invoked in one of the following ways:

- Directly from the operating system shell (for example Solaris or Windows NT).
- By an automated script.

- Indirectly by a Common Gateway Interface (CGI), a standard mechanism that an HTTP server uses to execute programs external to the Web server itself.

Your LivePayment application or the **cpcmd** utility sends transaction information to the *card processor*, a continuously operating process which routes transaction information through the Internet to the banking network. The card processor implements SSL security and message transmission.

When the card processor routes information through the Internet, it uses the *bank card gateway*, which is located at the banking institution. Transaction information goes through the gateway to the *bank card acquirer* (a bank authorized to accept financial transactions on behalf of a merchant). The acquirer processes transactions, routing responses back to LivePayment. The response codes are available within the LiveWire script or program that initiated the payment transaction.

A payment-enabled Web site also includes an *administration server,* which the system administrator uses to install and administer LivePayment, LiveWire, and the HTTP server.

Processing credit cards with Netscape LivePayment

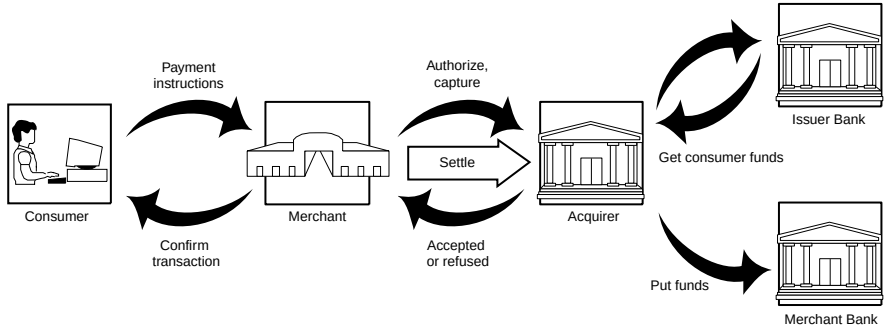
Making purchases over the Internet involves three parties: the consumer, or credit card holder, the merchant who is offering products or services for sale, and the financial institution that processes credit card payments, known as the acquirer. Indirectly, there are two other parties involved: the bank that issued the consumer's bank card, and the merchant bank, where the merchant's account resides.

- **The consumer** interacts with the merchant's Web site using a Web browser, such as Netscape Navigator. In order to make credit card purchases, the consumer must obtain a bank card from an issuer bank, and provide bank card information to the merchant's application when he or she decides to make a purchase.

- **The merchant** develops the application that makes goods and services available for sale. In order to accept credit card payments, the merchant must have an established relationship with a merchant bank and an acquirer.
- **The acquirer** is the financial institution that operates the payment gateway used to accept transactions from merchants on the Internet, on behalf of merchant banks. The acquirer and the merchant bank can be the same institution, or they can be separate institutions.
- **The issuer bank** is the bank that issued the bank card to the consumer.
- **The merchant bank** is the bank where the merchant's account is established. The merchant bank may also function as the acquirer, or it may designate another financial institution to function as acquirer on its behalf.

The following figure shows LivePayment's credit card transaction process.

Figure 1.2 Credit Card Transactions Using LivePayment



1. When the consumer decides to buy something, the Web application prompts the consumer for credit card information, usually along with other information such as shipping address.
2. The consumer enters payment information into an SSL-secured form. The payment information is sent to the merchant protected by SSL.
3. The merchant creates a *slip* (an encrypted, electronic analogy to a paper credit card slip) and sends the slip to the credit card acquirer for authorization via the LivePayment card processor. The information on the slip is used for the authorize, capture, and credit transactions.

4. The acquirer responds either with an *authorization* for a certain amount of money, or refuses the transaction.

5. Assuming the transaction is authorized, a *capture* is the next step. The capture takes the information from the successful authorization and charges the authorized amount of money to the credit card.

   Because the merchant should not capture until the ordered goods can be shipped, there may be a time lag between the authorization and the capture.

6. If a customer returns goods or cancels an order, the merchant generates a *credit* for the customer using the information on the original record of the purchase.

7. The final step is to *settle* the transactions between the merchant and the acquirer. Captures and credits usually accumulate into a *batch* and are settled as a group. This step effectively confirms all the transactions.

   The merchant and the acquirer compare the total sales amounts and number of sales transactions, and total credit amounts and number of credit transactions. Any discrepancies are worked out between the merchant and the acquirer. Settling can occur after every completed transaction, or as a group. This group of transactions is called a *batch*.

8. When the transactions are settled, the acquirer begins the transfer of money from the consumer accounts at various issuer banks to the merchant's account at the merchant bank.

Developing a payment processing application

LivePayment offers you three ways to develop a payment processing application:

- Modify the Starter Application Set provided with LivePayment.
- Create your own application with the LivePayment objects.
- Create your own application using the **cpcmd** utility.

You also have the option of combining these methods. For example, you might choose to have most of your application run in LiveWire, but use the utility to start and settle batches, since the utility can be more easily automated.

Modifying the Starter Application Set

LivePayment includes a Starter Application Set you can modify to quickly create your online commerce application. The Starter Application Set contains functions you can use to tailor the application to your business.

Before modifying the Starter Application Set, you should understand how card processing applications work. For more information, see Chapter 3, “Payment application concepts”.

For information on the Starter Application Set and how to modify it, see Part 2, “Using the LivePayment Starter Application Set.”

Creating your own application using LiveWire

In most cases, the Starter Application Set should be able to get you up in running with little modification. However, merchants with special business requirements (for example, support for multiple merchant environments) may want to create their own applications. Creating your own credit card application requires a thorough knowledge of credit card processing and how the LivePayment objects work. In addition, the Starter Application Set provides an example for you to look at. The following chapters will help you create your own application:

- Chapter 3, “Payment application concepts”
- Chapter 8, “Using the LivePayment objects”
- Chapter 9, “LivePayment object reference”

You should also read the *LiveWire Developer’s Guide* for instruction on developing with LiveWire.

Creating your own application using cpcmd

You can also create your own application using the **cpcmd** utility provided with LivePayment. The utility has several commands that you can use in a CGI program or operating system script to create a credit card processing application. It is very important that you understand the basics of setting up a credit card processing system. Before developing your application, be sure to read the following chapters:

- Chapter 3, “Payment application concepts”. Note that **cpcmd** (which does not encapsulate a relational database) requires you to manage your transaction states. Pay special attention to the information on transaction states in this chapter.
- Chapter 10, “Using the cpcmd utility”.

Certifying your application

After you have created your own application or modified the Starter Application Set, you may need to get it certified. Certification assures that your application meets credit card processing standards. You need your application certified under the following circumstances:

- If you create an application from scratch (without using the Starter Application Set).
- If you modify the Starter Application Set (LPStart and LPAdmin) by changing the .js files in the readonly_lib directory.

See “Certifying your application” on page 251 for more information.

Data storage

You can use a range of relational databases or flat file systems to store information from your LivePayment application. The sample application has been written to save transaction information into a relational database which you can use for your own application.

If you use LiveWire's database connectivity library you can also hook LivePayment into a variety of other popular relational databases. For more information on LiveWire and the databases it supports, see the LiveWire documentation.

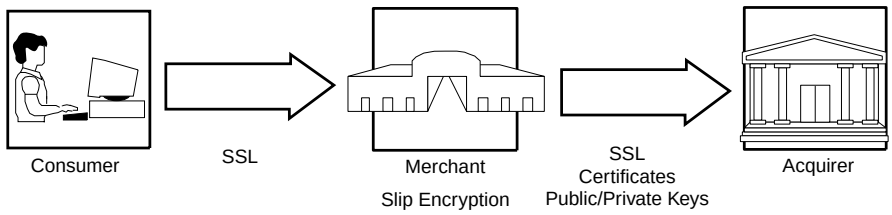
Applying for service from your bank

In order to process payment transactions, you need to set up a relationship with a bank card acquirer. For more information "Setting up a relationship with a bank card acquirer" on page 27.

Security

Credit card information is protected a number of ways as it travels through the system. It must be protected when it travels from the customer to the merchant over the Internet, at the merchant's site, and when it travels from the merchant to the acquirer. The following figure illustrates this process:

Figure 1.3 Credit Card Information Security



When credit card information travels over the Internet, it is protected by a secure transfer protocol—Secure Sockets Layer (SSL). The customer must use a Web browser such as Netscape Navigator that supports SSL to connect to the merchant's Web site. SSL protects the information as it travels from the customer to the merchant's Web site. The merchant must be running a secure server in order to ensure private communication.

Once the merchant receives the credit card number the merchant saves it into a slip and encodes the information. After the slip is encoded, most of the information on the slip (including the credit card number) is unreadable by the merchant. The merchant can store the encrypted slip in the database so there is a record of the transaction, but the sensitive information is shielded.

From the merchant's site, the encrypted credit card information travels over the Internet to the acquirer. The transfer is protected by SSL using mutual authentication to ensure that the acquirer knows the merchant company is who they claim to be, and the merchant knows that the acquirer really is the acquirer.

Identity is proved by security certificates, which are issued by a known Certificate Authority. The Certificate Authority (CA) verifies the identity of the merchant and the acquirer and establishes a hierarchy of trust. The authentication mechanism, which is embedded in SSL, uses public/private key cryptography.

Public/private key cryptography has two keys, a public key and a private key.The *public key* is publicized as widely as possible. The *private key* is kept completely secret. The merchant sends information to the acquirer encrypted using the acquirer's public key. The acquirer decrypts it using the private key. No one but the acquirer can decrypt the information, since only the acquirer has the private key.

When the acquirer receives the information, it has been protected by SSL and slip encryption. SSL provides the authentication, privacy and message integrity required. Replies from the acquirer to the merchant are also sent using SSL and therefore also achieve the authentication, privacy and messages integrity required. The acquirer encrypts the message using the merchant's public key and so only the merchant can decrypt it. Replies from the merchant to the customer are protected by SSL.

Netscape's SSL uses cryptographic algorithms developed by RSA Data Security.

Before you can transfer payment information across the Internet with LivePayment, you must have a security certificate, which is issued from a Certificate Authority. For more information, see Chapter 2, "Setting up Netscape LivePayment".

Further reading on security

For information about securing your server, and for a discussion of security concepts, see your server documentation.

In addition, the following resources help you begin learning and understanding issues related to security. Some are platform-specific:

- http://home.netscape.com/info/security-doc.html
- http://www.rsa.com/
- http://www.verisign.com/
- http://www.cis.ohio-state.edu/hypertext/faq/usenet/security-faq/faq.html
- *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*. Bruce Schneier. John Wiley & Sons, Inc., 1996.

Usenet newsgroups that regularly discuss computer security include: comp.security.misc, comp.security.unix, and alt.security.

Chapter

2

Setting up Netscape LivePayment

This chapter contains information on setting up Netscape LivePayment after it has been installed. It contains information on setting up a relationship with a bank card acquirer, configuring Netscape LivePayment, configuring security, and administering Netscape LivePayment.

This chapter contains the following sections:

- Post-installation procedures
- Changing LivePayment operating modes
- Accessing the Netscape LivePayment page
- Configuring the LivePayment parameters
- Configuring the card processor parameters
- Configuring security
- Administering Netscape LivePayment
- Administering LiveWire

Post-installation procedures

When you install LivePayment, you install the default configuration. Using this configuration you can start developing a credit card payment application in loopback mode. However, in order to run "live" transactions, you need to follow these steps.

1. Apply for merchant authorization to a bank card acquirer.
2. Generate a key pair and apply for security certificate (if you want LivePayment to use a different key pair and certificate than your server).
3. Configure LivePayment.
4. Configure the card processor.
5. Install your security certificate, or if you are using your server's certificate, set up LivePayment to use the server's certificate.
6. Design your application. You can design your application by altering the Starter Application Set, by using the LivePayment objects, or by using the **cpcmd** utility.
7. Start the card processor.
8. Run your application in test mode.
9. Enter production mode.

It may take time to get authorization from a bank card acquirer to process transactions over the Internet. You should apply for it immediately after you install LivePayment. For more information, see "Setting up a relationship with a bank card acquirer" on page 27. If you need to apply for a security certificate from a certificate authority, you should also do that immediately. For more information, see "Configuring security" on page 35.

While you are completing these two processes, you can begin developing your application using default values. For more information on starting to develop an application, see Chapter 3, "Payment application concepts".

Setting up a relationship with a bank card acquirer

Before you can process transactions with LivePayment you need to set up a relationship with a bank card acquirer. The bank card acquirer is a bank authorized to accept credit card transactions on behalf of the merchant. The acquirer is a bridge between the merchant and the bank that issued the credit card.

Your bank card acquirer needs to provide you with the following information:

- Merchant number
- Terminal number
- Merchant source ID
- Hostname of the bank card gateway
- Port numbers of the bank card gateway

Netscape Communications can refer you to available acquirers set up to process your credit card transactions, but cannot guarantee your acceptance. Owning and using Netscape software does not guarantee you the privilege of conducting credit card business over the Internet. Your ability to accept credit cards as a form of payment is subject to the acquirer's approval only.

If your business is handling credit card transactions, you are required to have MOTO (Mail Order/Telephone Order) approval through a Netscape supported acquirer. If Netscape does not support the acquirer that you are MOTO approved with, you cannot conduct automated credit card transactions.

You must set up a business agreement with an acquirer that includes arranging for the settlement of credit card purchases between your bank and your customer's card issuing bank.

For a list of acquirers and how to contact them, see Appendix B, "Netscape-supported bank card acquirers".

Changing LivePayment operating modes

You can operate LivePayment in the following modes: loopback, test, and production. *Loopback mode* is the default mode when LivePayment is installed. It is for initial development purposes *only*. You cannot perform transactions through the bank card gateway in loopback mode. In *test mode* you test the gateway and your application with test values for the bank card gateway interface. No financial transactions are actually performed (no money is involved) but you can send test values to the acquirer and receive responses. *Production mode* is the "live" mode, in which you can perform actual payment transactions with real data.

The modes have different merchant and terminal numbers, port numbers, and gateways. You need to get all these values from your acquirer.

The following table summarizes the differences between the modes:

	Loopback Mode	Test Mode	Production Mode
Internet Access	Not required	Required	Required
Card Processor	Not running	Running	Running
Certificate	Not required	Required	Required
Gateway Host	Not required	Test gateway from the acquirer, for example: ccgwt.card.net	Permanent gateway from the acquirer, for example: ccgw.card.net
Port Number	Not required	Available from your acquirer	Available from your acquirer.
Merchant Number	0000000000	Temporary number from acquirer	Permanent number from acquirer
Terminal Number	0000000000	Temporary number from acquirer	Permanent number from acquirer

Starting up in loopback mode

Loopback mode is the default mode when you install LivePayment. You do not need to configure LivePayment to run in this mode. The merchant and terminal numbers are set to 0000000000. The gateway host and port number are not set to default values.

Note: Loopback mode is for initial development ONLY. You must change to production mode before you can actually run credit card transactions. Also, note that the AVS response is a random response when you run in loopback mode. The random response allows you test for all possible responses.

Changing from loopback mode to test mode

When you install LivePayment initially, the installer automatically configures LivePayment to perform in loopback mode. To change from loopback to test, follow these steps:

1. Obtain your test merchant and terminal numbers from your acquirer.
2. Change the LivePayment parameter configuration to *not* use loopback mode.
3. Update the LivePayment parameter configuration with the test values for the merchant number and the terminal number.
4. Update the card processor parameter configuration with the test values for the host and port number.
5. Configure security.
6. Start the card processor.
7. If you used a database in loopback mode (for example, if you ran the Starter Application Set) you need to clean that information out of your database before using test mode. You can also create a new database for running in test mode. Data produced in loopback mode cannot be settled in test mode.

You use test mode to verify your application. You must test your application's ability to perform the credit card transactions, check for errors, and send and receive information through the gateway.

Changing from test mode to production mode

When you have thoroughly tested your application, and are ready to move to production mode, follow these steps:

1. Shut down the card processor.
2. Make sure the LivePayment parameter configuration is set to *not* use loopback mode.
3. Update the card processor parameter configuration with the production values for the merchant number and the terminal number.
4. Update the card processor parameter configuration with the production values for the host and port number.
5. Restart the card processor.

In addition, when you change from test mode to production mode you might want to change the password in the file that's used to encode and decode the slips. The password in the password file that is shipped with LivePayment is not unique, so you should change it. However, after you change the password, you cannot decode slips that were encoded with the old password. Choose your time to change the password carefully.

Accessing the Netscape LivePayment page

To configure and administer Netscape LivePayment, access the Netscape LivePayment page. To access it, follow these steps.

1. **Click the LivePayment link from the Sever Selector page or enter the URL to the LivePayment page.**

   Access the Netscape LivePayment page through the Administration server. Use the URL http://*server_name:PORT*, and click the server ID under Netscape Livepayment.

   Or, you can enter the full URL path into the browser.

   `http://server_name:PORT/livepayment-ID/bin/index`

   where *server_name* is the name of your server, *PORT* is the port number of your administration server, and *ID* is the ID of your server.

2. **Enter your login and password.**

   A pop-up box is displayed. Enter your server administration user login and password.

   The Netscape LivePayment page is displayed:

   This page has two content frames. One is a menu of the LivePayment configuration and administration forms. The other frame displays the LivePayment forms themselves. By default, the Administer Card Processor form is displayed.

   To return to the Server Selector page, click the **Server Selector** link.

Configuring the LivePayment parameters

The LivePayment parameters are values that your application can pick up by default. They also contain information on your current LivePayment operating mode. The values you can set are:

- Merchant name
- Merchant number
- Terminal number
- Password file

In addition, you can also set whether or not you are using loopback mode.

To set these values, follow these steps.

1. **Display the Configure LivePayment Parameters form.**

   From the Netscape LivePayment page, click the **Configure LivePayment Parameters** link to display the Configure LivePayment Parameters form.

2. **Fill in the parameter configuration information.**

   You can enter values for the following parameters:

   Merchant Name is the name of the merchant doing business on the Internet.

Merchant Number is the merchant number given to you by your acquirer. This number can be the default value (all zeros) that is automatically set during installation, which means you are running in loopback mode. It can also be the temporary test number if you are running in test mode, or the permanent number if you are running in production mode.

Terminal Number is the terminal number given to you by your acquirer. This number can be the default value (all zeros) that is automatically set during installation, which means you are running in loopback mode. It can also be the temporary test number if you are running in test mode, or the permanent number if you are running in production mode.

Password File is the path to the file that contains the password that encodes and decodes the slips. LivePayment ships with a default value for the password. For security reasons you should change this password before entering production mode. However, once you change the password, you cannot decode any slips that were encoded with the previous password.

Use Loopback Mode designates whether or not you are running LivePayment in loopback mode. If the field is set to YES, you are running LivePayment in loopback mode. If it is set to NO, you are running in test or production mode.

3. **Save your changes.**

   Press the **OK** button to save your changes.

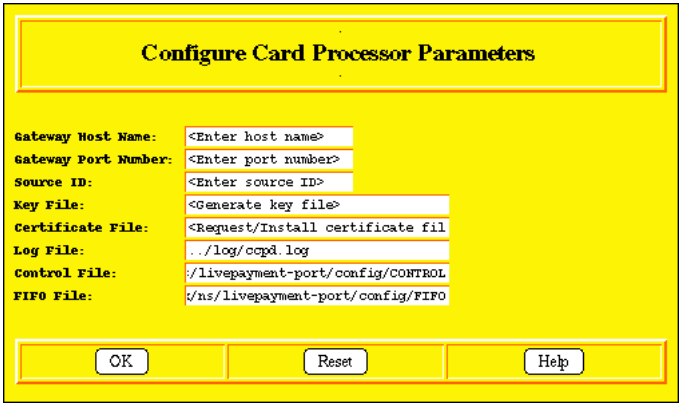
Configuring the card processor parameters

The card processor requires parameters to transmit information across the Internet to the acquirer. Some of these fields are provided by the acquirer when you establish service. If you do not have these values yet, you can run in loopback mode. For more information on loopback mode, see “Changing LivePayment operating modes” on page 28.

To configure the card processor, follow these steps.

1. **Display the Configure Card Processor Parameters form.**

   On the Netscape LivePayment page, click the **Configure Card Processor Parameters** link. The Configure Card Processor Parameters form is displayed.



2. **Configure the card processor.**

   The fields on this form govern the transfer of payment data across the Internet. Some of these field values are provided by your bank card acquirer.

   Gateway Host Name is the hostname of the bank card gateway, which is provided by the acquirer. The card processor forwards all credit card transaction requests to this computer. The hostname differs depending upon whether you are running in test mode or production mode.

   Gateway Port Number is the port number of the bank card gateway, which is provided by the acquirer. The bank card gateway listens on this port number for requests from the card processor. The host port number differs depending upon whether you are running in test mode or production mode.

   Source ID is the merchant source ID provided by the acquirer.

   Key File is the location of your previously generated key pair file. The key pair file contains the public and private keys the card processor uses during secure communications with the gateway.

   Certificate File is the location of your certificate file. The certificate file contains the signed certificate that binds to the public key the card processor uses during secure communications with the gateway.

Log File is the path to the file that collects log data for the card processor. The path specified in the default is relative to the directory where the card processor was installed. You can accept the default value.

Control File is the path to a file that contains control information about your card processor. The file is created when you start the card processor. You can accept the default value.

FIFO File is the name of the communication pipe between the application and the card processor. You can accept the default value.

3. **Save the changes.**

   Click the **OK** button to change the parameter values.

Domain Name Service (DNS) aliasing

The card processor delivered with LivePayment can work with acquirers who have configured their gateways to use Domain Name Service (DNS) aliasing. This allows for a given logical hostname to map to multiple IP addresses. DNS lookup spreads the load of multiple card processors connecting through the same gateway.

For example, the logical hostname of the FDC bank card gateway is ccgw.card.net. This logical hostname is mapped (by the DNS) to two different IP addresses: 165.90.142.2 and 204.254.78.2.

At startup, the card processor queries the domain name server with the given gateway hostname to retrieve a list of all possible IP addresses. From the list of addresses, the card processor randomly picks one address at a time and attempts to connect to that gateway. It stops at the first successful connection. The card processor reports an error and exits if it cannot connect to any of the addresses.

Configuring security

You have two options for configuring security for LivePayment. You can either use the same key pair file and certificate that the server uses, or you can have a separate key pair and certificate for LivePayment. Having two separate certificates gives you a higher level of security.

Using the server's certificate

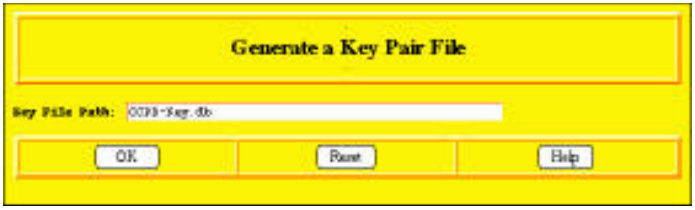
To use the server's key pair file and certificate, you have to copy the key file from the server directory to the LivePayment configuration directory. You must also have kept the original certificate email from the CA.

Copying the key file

To copy the key file to the LivePayment configuration directory, follow these steps:

1. **Display the Generate a Key File form.**

   On the Netscape LivePayment page, click the **Generate a Key File** link. A help page and the following form appear:



2. **Copy the key file from the server directory to the LivePayment configuration directory.**

   Ignore the instructions in the help page. Instead, copy your key file from the server directory to the LivePayment configuration directory. For example, copy the key file:

   server_dir/https-*ID*/config/ServerKey.db

   to the LivePayment configuration directory, for example:

   server_dir/livepayment-*ID*/config/CCPD-Key.db

   where *server_dir* represents the directory where the server is installed, and *ID* is the server identifier of your HTTP server.

3. **Update the file path.**

   If you do not specify a path name, the default is the LivePayment configuration directory. If you want to store CCPD-Kye.db in a different directory, specify the path to that directory in the Key File Path field.

4. **Save your changes.**

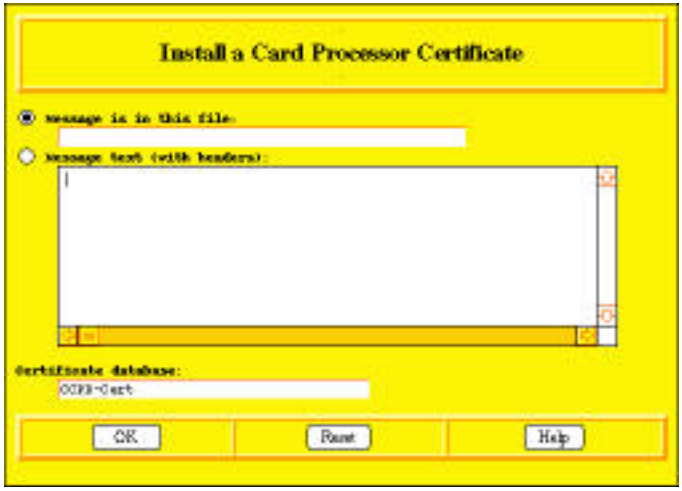
   Click **OK**.

Copying the certificate

To use the server's certificate, follow these steps:

1. **Display the Install a Card Processor Certificate form.**

   From the Netscape LivePayment page, click the **Install a Certificate** link.



   The email you received from the CA contains the certificate. You either need to specify the file name where the entire message was saved or cut and paste the message text into the box provided.

2. **Specify a destination directory for the certificate.**

   Specify the path to the certificate file. This file should not appear in your document root directory or any generally available directory. If you do not provide a path, the default is the LivePayment configuration directory.

3. **Save your changes.**

   Click **OK**. The server extracts the certificate from the email and saves it to the directory you specified.

Using a separate certificate for LivePayment

If LivePayment has its own key pair and certificate, use the following procedure:

1. Generate a key pair.
2. Request a certificate from a Certificate Authority.
3. Install the certificate when it is transmitted back to you from the Certificate Authority.

Generating a key pair file

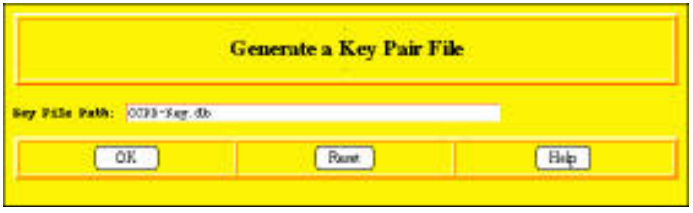
You need to generate a key pair file that holds the *public* and *private* keys for LivePayment's card processor. These keys are used during secure communications between the card processor and the bank card gateway. The private key is stored in encrypted form using a password you specify.

- A public key is usually used to exchange session keys. It is also used to verify the authenticity of digital signatures and to encrypt data.
- A private key is usually used to decrypt session keys. You always keep your private key secure. The server key file password protects the key, but for additional security you shouldn't keep the key file in a directory where people have access to it. The private key is also used to create a digital signature when you first request a certificate.

To configure security, follow these steps:

1. **Display the Generate a Key Pair File page.**

   On the Netscape LivePayment page, click the **Generate a Key File** link. The Generate a Key Pair File form appears.



The help window containing instructions for generating a key pair file automatically opens.

2. **Open a new window beside the browser window.**

3. **Log on as the server user.**

4. **Change your directory to the server root.**

5. **Run the key file generation program.**

   The program is in the bin directory of your LivePayment directory. Type:

   ```
   bin/livepayment/admin/bin/sec-key
   ```

6. **Type a location for the new key pair file.**

   Usually, the key pair file is stored in the server root, under the directory livepayment-*ID*/config, with the file name CCPD-Key.db. *ID* is the name of the server identifier. The directory you use should be safe from other users. For example, use a directory that only the server has read and write access to.

7. **Generate the key pair.**

   A screen with a progress meter appears. Depending upon your operating system, you either move the mouse randomly or type random keys at different speeds until the progress meter is full.

   The randomness is used to create a unique key pair file.

8. **Type in a password for your key pair.**

   Any time the card processor is restarted, you must type the password to decrypt the key file and extract the public and private keys.

   The password must be at least eight characters in length. It is required that the password have at least one non-alphabetical character (a number or punctuation mark) somewhere in the middle. Make sure you memorize this password. The security of your card processor is only as good as the security of the key file and its password.

9. **Confirm the password.**

   Retype the password and click **OK**.

10. **Return to the Generate a Key Pair File page.**

    Click the browser window displaying the Generate a Key Pair File page.

11. **Enter the path to the key file.**

    In the **Key File Path** field, type the path and file name of the key file. This directory should be safe from other users. For example, use a directory that only the server has read and write access to. If no directory is specified, the LivePayment configuration directory is the default.

12. **Save your changes.**

    Click **OK**. The system generates the key-pair file and places it in the directory you specified.

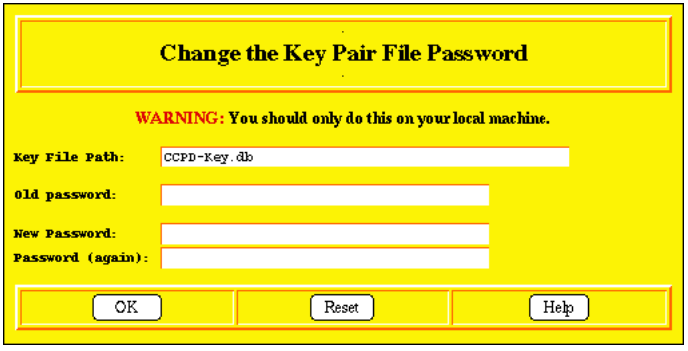
Changing the key pair file password

Periodically you may want to change your key pair file password for security reasons. If your server is not already running in secure mode, you must submit this form from the actual server machine or over a trusted network. Otherwise, your password could be intercepted over the network.

To change the password, follow these steps:

1. **Display the Change the Key Pair File Password form.**

   From the Netscape LivePayment page, click the **Change Key Password** link.



2. **Enter the path to the key file.**

3. **Enter the old password.**

4. **Enter the new password.**

   This password must be at least eight characters long and contain at least one non-alphabetical character (a number or a punctuation mark) somewhere in the middle.

5. **Confirm the new password by entering it again.**

6. **Save the changes.**

   Click the **OK** button to change the password.

Requesting a certificate

The *certificate* proves your identity before beginning a secure transaction. *Certificate Authorities (CAs)* are trusted third-party companies that can approve requests for signed digital certificates. Note that not everyone who requests a certificate is given one. Also, it can take anywhere from a day to two months or more to approve a certificate. You are responsible for promptly providing all the necessary information to the CA.

Before you can request a certificate, you must choose a CA and contact them regarding the specific format of the information they require. When you purchased LivePayment, you received a list of CAs. Most CAs require that you prove your identity before they issue a certificate. For example, they want to verify your company name and who is authorized by the company to administer LivePayment and whether you have the legal right to use the information you provide.

To request a security certificate, follow these steps:

1. **Display the Request a Card Processor Certificate form.**

   To access the form, click the **Request a Certificate** link on the Netscape LivePayment page. The Request a Card Processor Certificate form is displayed.

2. **Fill in the information to send to the CA.**

   The form requires the following information:

   Certificate Authority is the email address of the CA you have chosen.

   New Certificate or Certificate Renewal indicates whether you are applying for a new certificate or a renewal. Many certificates expire after a set period of time, such as six months or a year. Some CAs will automatically send you a renewal.

   Key File Location is the location of your previously generated key pair file. The server uses this information to encrypt a message to the CA, and send the public key.

Key File Password is the password of your previously generated key pair file. The server uses this information to encrypt a message to the CA, and send the public key.

Common Name is usually the fully qualified hostname used in DNS lookups (for example, www.netscape.com). However, some CAs might require different information, so it's very important to contact them.

Email Address is your business email address. This is used for correspondence between you and the CA.

Organization is the official, *legal* name of your company, educational institution, partnership, and so on. Most CAs require that you verify this information with legal documents (such as a copy of a business license).

Organizational Unit is an optional field that describes an organization within your company. This can also be used to note a less formal company name (without the *Inc.*, *Corp.*, and so on).

Locality is an optional field that usually describes the city, principality, or country for the organization.

State or Province is usually required, but can be optional for some CAs. It cannot be an abbreviation.

Country is a required, two-character abbreviation of your country name (in ISO format). The country code for the United States is US.

Telephone Number is your phone number. Be sure to include your area code and any international codes as applicable. The CA uses this number to contact you regarding your request for a certificate.

3. **Send the request to the CA.**

   When you have filled out the required information, click **OK.** The server composes an email to the CA that includes your information. The email has a digital signature created with your private key. The digital signature is used by the CA to verify that the email wasn't tampered with during routing from your server machine to the CA. In the rare event that the email is tampered with, the CA will usually contact you by phone.

You can't continue configuring security until your request for a certificate is approved and a confirmation is sent to you by email.

In the certificate file, all the fields together are called the *distinguished name*. The distinguished name in a certificate is not seen by users but it aids in uniquely identifying certificates to programs that need to identify them.

Some CAs offer certificates that indicate a greater level of detail and veracity to vendors or individuals who provide greater proof of their identity. For example, you might be able to purchase a certificate that states that the CA has not only verified that you are the rightful administrator of the www.danishfurniture.com web site, but that you really are a furniture dealer, have been in business for ten years, and have no outstanding customer litigation against you. Generally, these certificates cost more than standard ones.

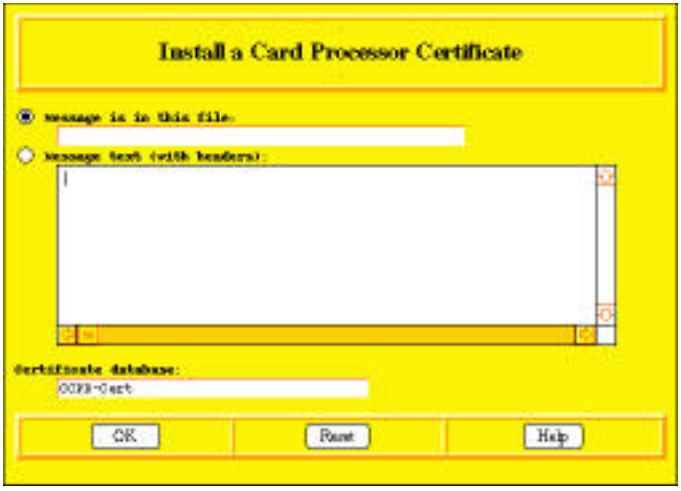
Installing a certificate

When you receive your certificate via email from the CA, it will be encrypted with your public key so that only you can decrypt it. The server will decrypt it when you install it. You can either save the email somewhere accessible to the server or copy the text of the email and be ready to paste the text into the Install a Card Processor Certificate form.

To install the certificate, follow these steps:

1. **Display the Install a Card Processor Certificate form.**

   From the Netscape LivePayment page, click the **Install a Certificate** link.



   The email you received from the CA contains the certificate. You either need to specify the file name where the entire message was saved or cut and paste the message text into the box provided.

2. **Specify a destination directory for the certificate.**

   Specify the path to the certificate file. This file should not appear in your document root directory or any generally available directory. The default directory is the LivePayment configuration directory.

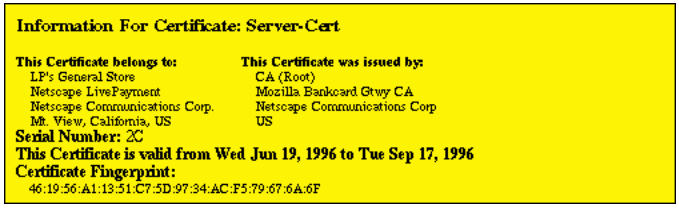
3. **Save your changes.**

   Click **OK**. The server extracts the certificate from the email and saves it to the directory you specified.

Examining a certificate

After your certificate is installed, you may need to examine it. For example, you might need to find out when it expires, or to provide certificate information to an acquirer.

To examine a certificate, on the Netscape LivePayment page click the **Examine a Certificate** link. The certificate information is displayed.



Administering Netscape LivePayment

Once you have installed LivePayment, obtained your certificate, and set up your relationship with your acquirer, you perform the following administrative tasks:

- Start the card processor
- Check the card processor's network connection
- Stop the card processor

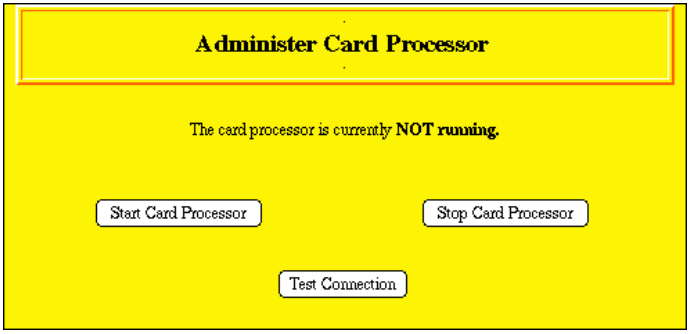
You can also administer the LiveWire development environment from within LivePayment. For more information, see “Administering LiveWire” on page 47.

Starting the card processor

To start the card processor, follow these steps:

1. **Display the Administer Card Processor page.**

   On the Netscape LivePayment page, click the **Administer Card Processor** link. The Administer Card Processor page is displayed.



   The page indicates whether the card processor is running. To start the card processor, follow the directions on your screen. You will have to type random keystrokes and enter the password to your key pair file.

   When you have entered the password successfully, you exit the program. The card processor is running.

Checking the gateway connection

To test the network connection between the card processor and the acquirer through the bank card gateway, follow these steps:

1. **Display the Administer Card Processor page.**

   On the Netscape LivePayment page, click the **Administer Card Processor** link. The Administer Card Processor page is displayed.

   A section of the page has a button for testing the card processor gateway connection.

2. **Test the Gateway connection.**

   Click the **Test Connection** button to find out if the gateway connection is working. The system tests the card processor by sending a sample transaction through the gateway. A confirmation page displays the results of the test.

Stopping the card processor

To stop the card processor, follow these steps.

1. **Display the Administer Card Processor page.**

   On the Netscape LivePayment page, click the **Administer Card Processor** link. The Administer Card Processor page is displayed.

   The page shows whether the card processor is running or not.

2. **Stop the card processor.**

   Click the **Stop Card Processor** button to stop the card processor. The page indicates whether the shut down was successful.

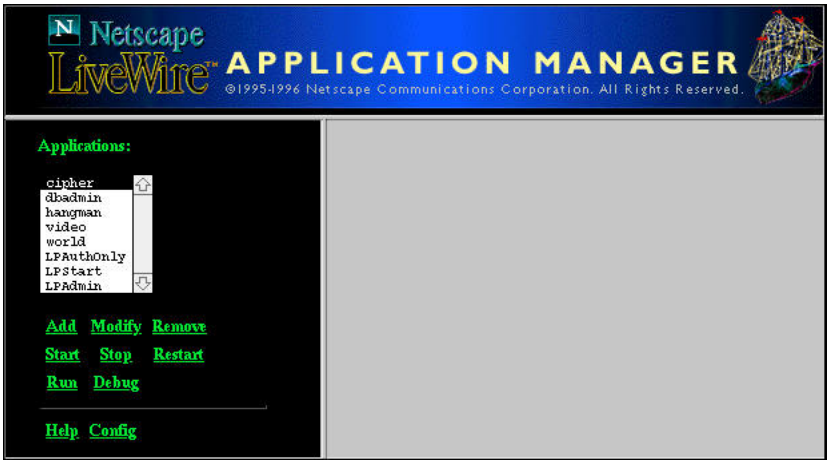
Administering LiveWire

The Netscape LivePayment page contains a link to the LiveWire Application Manager, where you can install, modify, restart, run and delete LiveWire applications on your server. Use the LiveWire Application Manager to administer the LIvePayment Starter Application Set and any LivePayment applications you create.

To use the LiveWire Application Manager, follow these steps:

1. **Display the LiveWire Application Manager page.**

   On the Netscape LivePayment page, click the **LiveWire Application Manager** link to access the LiveWire Application Manager. This page displays all currently installed LiveWire applications and allows you to install new ones.



2. **Display information about an application.**

   Highlight an application to see its information displayed in the frame on the right.

   Note that for LivePayment the external library is:

   /*server_dir*/bin/https/libccp.so

3. **Run an application**

   With the application you want to run highlighted, click **Run**.

For more information on this page, click the **Help** link, or see the LiveWire documentation.

Accessing the Server Selector

You can access the Server Selector page by clicking the Server Selector link on the LivePayment page. The Server Selector page contains links to use when administering your server. It also contains a link to the LivePayment page.

Chapter

3

Payment application concepts

This chapter includes information about developing with LivePayment, and describes the transaction flow for processing credit cards. Use the information in this chapter along with Chapter 8, “Using the LivePayment objects” and Chapter 10, “Using the cpcmd utility” to develop credit card applications.

This chapter has the following sections:

- Credit card transactions
- Business rules for credit card applications
- Designing your database
- Using batches
- Maintaining the payment and batch states
- Storing Information from the acquirer

Credit card transactions

Before writing your credit card application, you must understand the basic transactions for processing credit card information, and the communication between the card processor and the Bank Card Gateway.

The following sections describe the credit card transactions in detail. They refer to the three parties involved in credit card commerce on the Internet: the customer with the credit card, the merchant who has set up a store to conduct business on the Internet, and the bank card acquirer. The explanation assumes that the person creating the credit card application is the merchant.

Creating a slip

The slip mimics the paper charge slip created when a customer purchases an item at the store using a credit card. This information includes:

- Credit card number
- Amount allowed for purchases
- Expiration date
- Card type (for example, MasterCard, Visa etc.)

The slip also contains additional information:

- The cardholder's billing address (street address and zip code)
- The maximum amount that the merchant can charge against a slip.

The customer shops at a merchant's web site with a browser. If the customer decides to make a credit card purchase, the customer enters credit card and other information, which the merchant uses to create the slip.

The slip is encrypted for security reasons. Once encrypted, some of the credit card information is not accessible, for example, the card number and expiration date.

The information on the slip is used for authorizing and capturing a purchase.

Authorizing a purchase

After the customer requests a purchase and the merchant creates a slip, the merchant requests authorization of the purchase from the bank card acquirer. An authorization ensures that the customer has a valid credit card with enough credit for the purchase. When a purchase is authorized, the amount of money authorized is reserved against the card's credit limit. The amount of money is reserved for a period of time which is determined by the bank that issued the card.

The authorization information includes:

- Amount
- Currency
- Slip information

When the merchant authorizes a purchase, LivePayment sends authorization information to the card processor, which sends it through the Bank Card Gateway to the bank card acquirer.

The acquirer either rejects or approves the authorization. If the authorization is approved, the transaction can continue.

As part of the approval, the acquirer sends the merchant the authorization code, and may also send the payment service data and the address verification service result. The merchant saves this authorization information to use when capturing. See “Storing Information from the acquirer” on page 64 for more information on this acquirer-supplied information. In addition, the merchant should keep track of the state of the credit card transaction. See “Maintaining the payment and batch states” on page 65 for more information about the state.

Starting a new batch

A batch is a group of transactions that can be settled later as a group. Starting a new batch is sometimes called opening a batch. Some merchants start a new batch at the beginning of the business day and settle it at the end of the business day. Other merchants cycle through batches more or less often.

When the card processor starts up initially, or when a current batch is settled, transactions automatically start in a new batch. You need to know the batch number to capture a or credit an amount. The acquirer supplies the batch number. If you are using JavaScript, use **getCurrentBatch** to find out the latest batch number. You should store the batch number in the database.

See “Using batches” on page 58 for more information about batches.

Capturing a purchase

A lag time may occur between authorization (when the product is ordered), and capture (when the product is shipped). Information required for capture includes:

- Amount
- Slip information
- Batch number
- Event ID
- Values supplied by the acquirer at authorization

When the merchant is ready to ship the product he or she captures the purchase. The LiveWire or CGI program sends the capture information to the card processor, which sends it through the Bank Card Gateway to the bank card acquirer. The acquirer either rejects or approves the capture.

The merchant must maintain the state when the capture is sent to the acquirer and update it when the acquirer approves the capture. See “Maintaining the payment and batch states” on page 65 for more information on the state.

Issuing credit for a purchase

To credit the account of a customer for a purchase, the merchant must use information on the original record of the purchase and issue credit for the amount that was previously captured. The merchant does not need to perform an authorization for a credit.

The merchant maintains the state when the credit is sent to the acquirer and updates it when the acquirer approves the credit. See “Maintaining the payment and batch states” on page 65 for more information on the state.

Settling a batch

The batch continues to accumulate capture and credit data until it is settled. When settling a batch, the merchant checks the total number of sales and the total sales amount against the numbers recorded in the batch. The merchant also checks the total number of credits (refunds) and total credit amount. The merchant must track these values in the application. If there is a discrepancy, the merchant must contact the acquirer and resolve it.

The batch number is required to settle the batch. See “Using batches” on page 58 for more information about batches.

Business rules for credit card applications

When writing a credit card application, you must follow a few basic business rules about credit card processing.

Authorize

The following rules govern the use of **authorize**.

- You must authorize for at least the amount you are going to capture. If you authorize for less than the capture amount, you must perform another authorization. Some applications do an initial authorization of a small amount (for example, $1.00) to check that the card is valid, then when the product is ready to ship, authorize for the full amount.
- If you authorize for more than the amount you capture, you should notify your acquirer. You can still use the same authorization code.
- When a payment is authorized, the acquirer returns the AVS (address verification service) result. This field verifies the billing address given by the customer against the cardholder’s billing address. The AVS is a security

check to help ensure that the person using the card is actually the cardholder. At least one part of the response (the street address or the zip code) should match, otherwise you should refuse the transaction. If the acquirer does not check the AVS, or if the AVS does not match, you may still accept the transaction, but it presents a higher level of risk. For more information, see “Results of authorize” on page 64.

- Once a payment is authorized, the authorization remains active for an amount of time that is determined by the bank that issued the card. You must capture the amount during this period, or you have to authorize the payment again.

Capture

The following rules govern your use of **capture**:

- Though you can authorize when an order is made, you should not capture until the goods are shipped.
- You must take steps to ensure that a customer is charged only once for each purchase. For more information, see “Maintaining the payment and batch states” on page 65.
- You must have the current batch number.
- If your acquirer is FDC, your eventID or transactionID must be unique within the batch. If you do not use a unique ID, you may not be able to settle the batch later.

Credit

The following rules govern you use of **credit**:

- You are not required to run **authorize** before crediting an account. However, the card to which you are crediting money must be valid.
- You must have the current batch number.

- If your acquirer is FDC, your eventID or transactionID must be unique within the batch. If you do not use a unique ID, your transaction is ignored. A credit needs a unique ID—do NOT use the capture's ID.
- If your acquirer is FDC, you will not receive an immediate error under the following conditions when crediting. However, you should institute checks for the following conditions, since they could cause problems later.
  - FDC does not verify that the credit card numbers are correct for a credit card type when crediting. For example, if the credit card type is Visa, but the credit card number is a valid American Express number, the credit operation will still be successful. If you want to make this check, you should build a check of the card type against the type of the card number into your application.
  - FDC does not check the expiration date of the card when crediting. If the account number is valid, the credit is successful, even if the card has expired. You should include a check of the expiration date in your application before crediting.

Settle

The following rules govern your use of **settle**:

- Once you settle, all new transactions go into the next batch.
- While you are settling a batch, you cannot capture.
- You totals for captures and credits must match those of your bank card acquirer when you settle. If they do not match, you must contact your bank card acquirer.

Designing your database

Before you create a credit card database application, you should design, create, and populate (at least in prototype form) your database. If you are using LiveWire, you should read the information on developing a LiveWire application with a database in the *LiveWire Developer's Guide.*

You should store slip and payment information in your database. For information on what to store, see the documentation on the LivePayment objects, which includes a detailed explanation of what parameters the objects and their methods require. You also need to store information received from the acquirer as a result of transactions. For more information, see “Storing Information from the acquirer” on page 64.

Finally, you need to maintain and store the payment and batch states. For more information see “Maintaining the payment and batch states” on page 65.

Using batches

A batch includes the captures and credits as they happen, so that they can be settled later as a group. Some merchants, for example, start using a new batch at the beginning of the business day and settle it at the end of the business day.

Using batches in an application

This section summarizes the steps that must be included in an application for maintaining batches. You need the batch number when you capture a payment, credit an account, and settle a batch.

Note that when using JavaScript, we recommend getting the current batch number initially using the **getCurrentBatch** method, and storing the batch number in the database. After that, use the batch number stored in the database to create a new batch object using the **new** operator. In addition, when using JavaScript this procedure assumes that you will need to recreate the batch object before using it to capture or settle. The procedure also assumes that capturing and settling take place in different JavaScript sessions.

Getting and storing the batch number

You must find out and store the batch number before capturing or settling payments.

1. Find out the current batch number (for JavaScript use **getCurrentBatch**).
2. Write the current batch number to the database.

Using the batch number for capture

To capture, you need the batch, and if you're using JavaScript, you need to recreate the batch object.

1. Before capturing, retrieve the batch number from the database.
2. If using JavaScript, create a new batch object (using the **new** operator) so that you can pass it to the capture method.
3. Capture the payment.
4. Write to the database.

Using the batch number for credit

To credit a account, you need the batch, and if you're using JavaScript, you need to recreate the batch object.

1. Before crediting, retrieve the batch number from the database.
2. If using JavaScript, create a new batch object (using the **new** operator) so that you can pass it to the credit method.
3. Issue credit for the payment.
4. Write to the database.

Using the batch number for settle

You also need the batch to settle, and if you're using JavaScript, you need to recreate the batch object.

1. Before settling, retrieve the batch number from the database.
2. If using JavaScript, create a batch object (using the **new** operator) so that you can pass it to the **settleBatch** method.
3. Get the total sales amounts, total sales counts, total credit amounts and total credit counts from the database.

4. Settle the batch (if using JavaScript, use **settleBatch**).

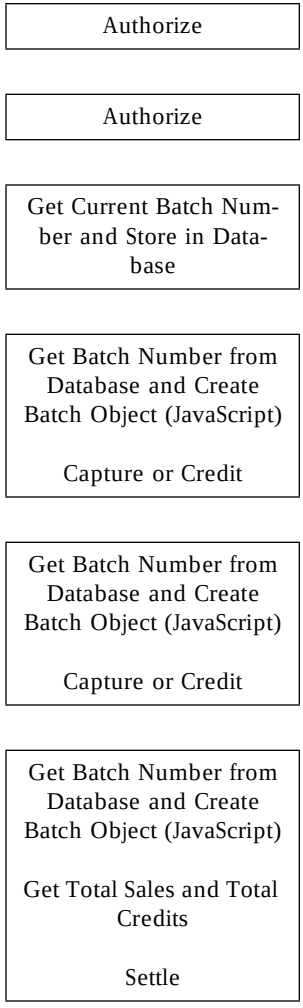
5. Write to the database.

Order of transactions

The order of transactions depends upon your business model—when the customer places an order, when you ship your product, and how many batches you use during a day.

Authorizations are not included in batches. Technically, you do not need to know the batch number until you start capturing. In addition, you might perform a number of authorizations before performing any captures.

For example, if a company ships “hard” goods and does not keep an accurate online inventory system, the flow might be as follows. Each box represents a different session.This chart only shows two authorizations and captures, but there could be more or fewer in actuality.

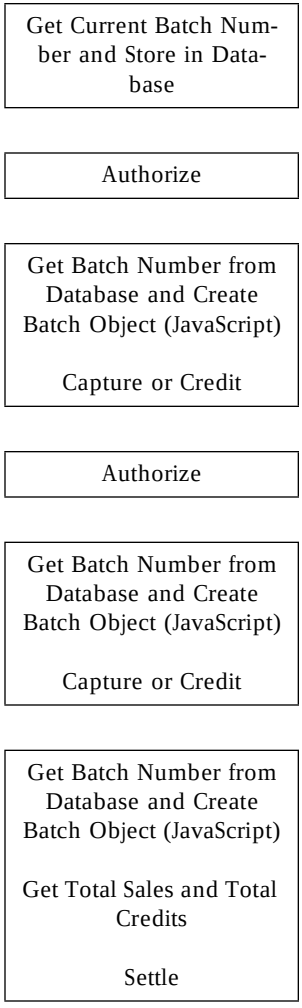


First, authorize the amounts. When you are ready to capture, you get the current batch and create the batch object. Then you can capture the amounts.

When you are ready to settle, you again get the batch number and create the batch object.

Authorizing purchases first and delaying captures until right before settling is a good model for “hard” goods. Because there may be a lag time between the authorization of a purchase and the capture, all captures are done immediately before settling. This model does not work as well for companies that have a high transaction volume, because many captures need to be done at once.

The following flow of transactions might be good for companies that sell “soft” goods, for example, an online publishing system that delivers information as soon as the customer enters a credit card number. This chart only shows two authorizations and captures, but there could be more or fewer in actuality.



First, you get the current batch. Then you authorize. Then you create a batch object. Then you capture. The process repeats until you are ready to settle. There is no delay between the authorize and capture, since there is no time lag between ordering a product and shipping it.

Storing Information from the acquirer

This section contains information on saving information provided by the acquirer as the result of credit card transactions.

You need to save the following information when it is sent to the application from the acquirer. All of this data is needed by later transactions. You can store the data in the database of your choice, for example, the database included in LiveWire Pro.

Results of authorize

When an authorization is successful, you receive the following information from your acquirer:

- Authorization code
- Payment Service Data
- Address Verification Service (AVS) Result

You need to save all of these fields in a database, for use later in capturing.

The authorization code is the acquirer's identification of the authorization. The Payment Service Data (also called the Interchange Compliant Code) is also provided by the acquirer when the authorization is approved.

The Address Verification Service is a three-character response that informs the merchant whether the address (billing street, billing zip code) given by the person ordering the goods matches the cardholder's address. The AVS is a security check to help ensure that the person using the card is actually the cardholder. At least one part of the response (the street address or the zip code) should match, otherwise you should refuse the transaction. Note that the acquirer will still authorize a payment even if the AVS does not match. It is up to the merchant to decide whether or not to continue with the transaction.

The AVS response is three characters. The first character represents whether the address matches, the second character represents whether the zip code matches, and the third character is the authorizer verification result code. “Y” is

a match, “N” is no match, and “X” is unavailable or incomplete service. If you receive a response of “X” you should continue with the transaction. You cannot interpret the Authorizer Verification Result code (the third character).

Address Match	Zip Code Match	Authorizer Verification Result Code
Y = Yes it matches.	Y = Yes it matches.	Opaque. You cannot interpret it.
N = No it does not match.	N = No it does not match.	Opaque. You cannot interpret it.
X = Match not checked or service incomplete.	X = Match not checked or service incomplete.	Opaque. You cannot interpret it.

Note: The AVS response is a random response when you run in loopback mode. The random response allows you test for all possible responses.

Batch number

The current batch number is provided by the acquirer when you query the system for the batch number (**getCurrentBatch** in JavaScript). You need the batch number to capture payments, as well as to settle. You must store the batch number when you get it from the acquirer. For more information, see “Using batches” on page 58.

Maintaining the payment and batch states

Each credit card transaction must have a *state* to track where the payment is in the transaction flow, and whether a batch is a current batch, or a settled batch. Because LivePayment does not maintain a state in a database of its own, you should maintain the state in the application you write. You can store the state in the database of your choice, for example, the database included in LiveWire Pro.

Why is the state needed?

You need to maintain the state for the following reasons:

- To keep track of where your payments are. For example, when you maintain the state you know whether a payment has been sent to the acquirer for capture or not.
- To ensure that the transfer of funds is done correctly. For example, when you maintain the state you know that a payment amount has already been captured, and you can avoid capturing it more than once. If the system should go down, the state provides a way to check up on every ongoing transaction.
- To keep track of whether a batch is the current batch or a settled, historical batch.

Keeping track of the payment states

The state field is a safety measure; by using it you know at any given time the payment's state. Netscape strongly recommends that you maintain states in your application. The following information provides guidelines for you in setting up your states.

The payment state typically changes at each point that information is sent to or received from the acquirer. For example, before you send a capture transaction through the card processor to the acquirer, the state changes to *capturing*. When the acquirer replies with transaction approval, the state changes to *captured*.

The following table shows the state for each payment transaction:

Payment Transaction	State
The payment has not been authorized yet.	none (not stored in the database)
The payment is about to be sent to the acquirer for authorization.	authorizing
The payment has been authorized by the acquirer.	authorized

Payment Transaction	State
The payment was authorized, but the purchase is rejected by the merchant (for example, if the AVS response indicates that the address does not match).	cancelled
The payment is about to be sent to the acquirer for capture.	capturing
The payment has been captured by the acquirer.	captured
The credit is about to be sent to the acquirer	crediting
The credit has been approved by the acquirer	credited

The state is most important for capturing and crediting, since those transactions are the ones in which money is transferred. However, it is a good idea to maintain the state for all transactions.

For example, the merchant sends a payment to the acquirer and changes the state to *capturing*. If the system goes down, the merchant will need to know the *capturing* state so that the merchant can contact the bank card acquirer and find out if the capture has happened on the acquirer's side. If the merchant does not track the state accurately, the customer may inadvertently be charged twice for the same order.

Keeping track of the batch states

The merchant must also maintain a state for your batches. The following information provides guidelines for you in setting up your states.

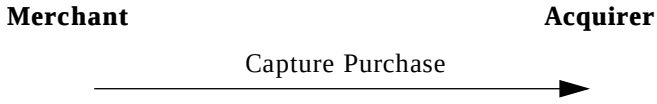
Batch Transaction	State
The batch has not been started.	none (not in the database)
The batch has been started (get current batch).	opened
The batch is being settled.	settling
The batch is settled.	settled

When you maintain the batch state you know whether a batch is currently receiving transactions (open). You also know when a batch is being settled. No captures should be performed while a batch is being settled. Once a batch has been settled, no more transactions are included in it. It is used for history data.

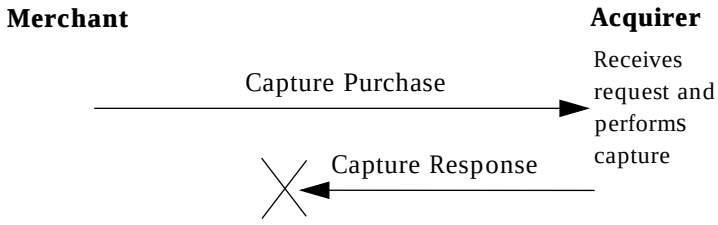
Idempotent transactions

Depending on your acquirer, some credit card transactions are *idempotent*. Idempotent transactions can be sent to the acquirer more than once, without resulting in a double transaction against the credit card. If your acquirer is First Data Corporation (FDC) the **getCurrentBatch**, **capture**, and **credit** transactions are idempotent.

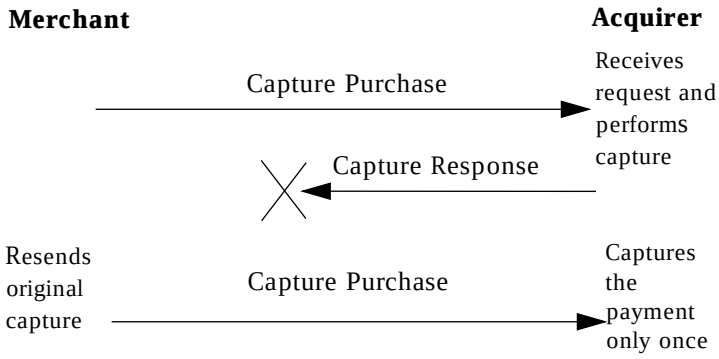
For example, a merchant sends a payment to the acquirer.



The acquirer receives it, performs a capture, and sends a capture response back to the merchant. However, the capture response never makes it to the merchant, for example, if the merchant's system goes down while the response is in transit.



When the merchant reboots the system, the merchant cannot tell whether the capture has been processed by the acquirer or not, because the response never arrived. Because capture is idempotent, the merchant can send the same capture again to the acquirer, and the acquirer will only charge the cardholder for one capture.



The following table shows which transactions are idempotent for FDC:

Idempotent	Not Idempotent
getCurrentBatch	authorize
capture	
settleBatch	
credit	

Netscape strongly recommends that you maintain the state, even if you also take advantage of the idempotent transactions in your implementation.

2

Using the LivePayment Starter Application Set

- **Running the Starter Application Set**
- **Reviewing the application code**
- **Modifying the Starter Application Set**
- **Netscape LivePayment Starter Application Set Reference**

Chapter

4

Running the Starter Application Set

The Starter Application Set is a set of applications that you can use to process credit card payments. You can use the Starter Application Set as a basis for payment-enabling your existing web site or developing a new site.

If you wish to review the application code, see Chapter 5, “Reviewing the application code.”

This chapter presents two simple scenarios, the first from the point of view of a customer running the application, and the second from the point of view of a merchant administering the application.

This chapter presents the following topics:

- Before you begin
- Starting the application
- Making a purchase
- Using the administration interface

The application files

The Starter Application Set is comprised of two LiveWire applications, LPStart and LPAdmin. LPStart is a simple payment processing application seen by customers when they purchase a product from you. LPAdmin is an application you use to administer the transaction data generated by customer purchases. In building your application, you can use LPAdmin "as is", but you'll need to modify LPStart.

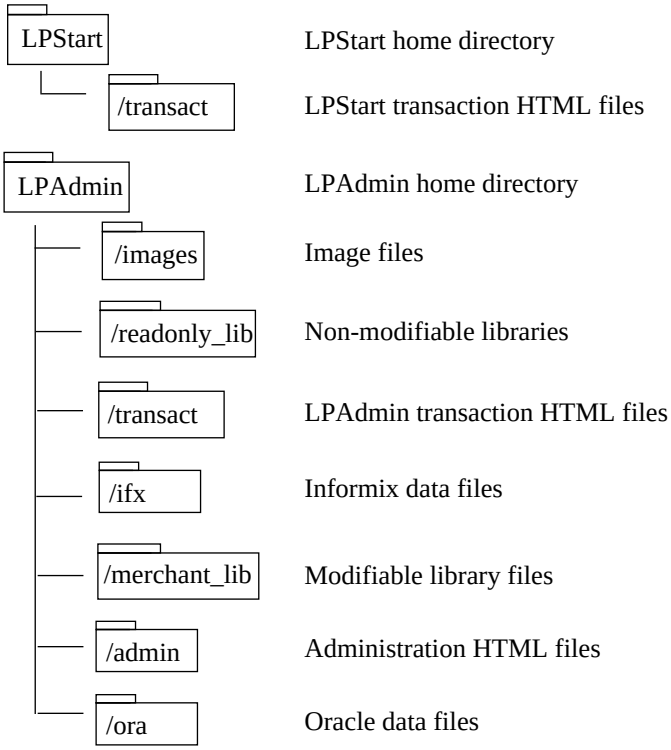
The application files are comprised of HTML files containing HTML code and .js libraries containing the JavaScript functions, the application code. The HTML files call the functions in the .js libraries.

As Figure 4.1 shows, the .js files are divided into two main directories, readonly_lib and merchant_lib. The readonly_lib directory contains the credit card processing libraries. The merchant_lib directory contains modifiable libraries. In general, you should not modify the code in the readonly_lib directory, and you should modify the code in the merchant_lib directory only after making backups and understanding the code.

The application code has been written generically and for reuse. In the majority of cases you will not need to modify the *merchant_lib* library code.

There are a number of HTML files required as form handlers. Currently, the HTML form pages cannot make direct calls to functions in the .js files. They must go through HTML form handlers.

Figure 4.1 Starter Application Set directories



Before you begin

Before running the Starter Application Set, you should have satisfied the prerequisites and configured the application to run, as described in the following sections.

Prerequisites

To run the Starter Application Set, you need:

- An Enterprise Server running.
- A relational database installed and running.
- LiveWire installed and enabled. Test the “video store” application to ensure LiveWire is properly configured and communicating with a database.
- LivePayment installed and running.

Before running the Starter Application Set, read Chapter 2, “Setting up Netscape LivePayment” and Chapter 3, “Payment application concepts”.

Configuration

The Starter Application Set needs to be configured to work in your environment. You need to modify the line in the start.html file that calls the **database.connect** method, which connects the Starter Application Set to the database. Modify the following properties in the start.html file for both LPStart and LPAdmin.

project.dbType— your database type (for example, "INFORMIX")

project.dbServer—your database server name

project.dbUsername— your database login name

project.dbPassword—your database password

project.dbName—the name of your database. Change this to something other than "LIVEPAYMENT." This should have the same value for LPAdmin and LPStart.

Starting the application

From the LiveWire application manager, run LPStart. The first page invites you to click on the **Purchase** link to purchase an item.

Making a purchase

1. **Click on the Purchase link to begin the purchasing process.**

   You are presented with a selection of three items: a coffee mug, a mouse pad, and a stuffed Mozilla. In this application, you may select only one item per purchase. Select any item by clicking on the radio button below it.

2. **Select an item to purchase and click on the Purchase Selected Item button.**

3. **Enter your billing and shipping information in the order form.**

   This includes your full name and your credit card type, number, and expiration date. Enter the shipping information. Clicking the **Clear Form** button clears all the information you've entered and lets your start over.

4. **Click the Complete Purchase button.**

   The application performs some checks on the information you submitted, creates a payment slip, and attempts to authorize it.

   If successful, the application displays a receipt, indicating your purchase was authorized. If it cannot authorize, the application displays an error message.

5. **Click on the Return Home link.**

   This returns you to the home screen.

Using the administration interface

Once you have made a purchase, use the cash register administration pages provided by LPAdmin to view how the system has recorded the transaction.

1. **From the LiveWire Administration page, run LPAdmin.**

   The administration screen offers links to other screens wherein you can view transaction information and perform collection transactions.

Viewing uncaptured authorizations

2. **From the main Administration screen, click on the Capture button.**

   This screen displays a table that reflects the information the system has recorded to identify your authorized, but uncaptured, purchase. Your purchase will appear as a row in this table. The system has recorded the merchant, credit card type, authorization code, payment service, AVS data, event ID, event time, slip ID, batch ID, and status.

3. **Click on the transaction ID.**

   You are presented with a transaction summary screen with two buttons, **Capture Transaction** and **Back**. These are used to capture a single transaction or back up a screen.

Capturing a transaction

Capturing is usually performed in batches of transactions. However, to capture a single transaction, for example if the goods have just shipped, you capture the transaction from this screen.

1. **Click on the Capture Transaction button.**

   This displays a message confirming the transaction has been captured.

Viewing transactions in the current batch

1. **From the main Administration screen, click on the Current button.**

   This displays a list of transactions in the current batch. These are captured transactions which have not yet been settled.

Cancelling a transaction

A transaction can be cancelled if it has not yet been captured. After a capture, the transaction must be reversed by a credit.

1. **From the administration screen, click on the Cancel button.**

   This displays a list of transactions in the current batch.

2. **Click on the transaction ID.**

   You are presented with a transaction summary screen with two buttons, **Cancel Transaction** and **Back**. These are used to cancel a single transaction or back up a screen.

3. **Click on the Cancel Transaction button.**

Crediting a transaction by batch

Crediting a customer's credit card may be necessary if the goods are returned or the transaction needs to be cancelled after a capture. To credit a transaction:

1. **Click on the Credit button.**

   This displays a list of batches in the database.

2. **Click on the ID of the batch which contains the transaction to be credited.**

   This displays the list of transactions in the batch.

3. **Click on the ID of the transaction you wish to credit.**

   This displays a summary page with information about your transaction and a **Credit Transaction** button.

4. **Click on the Credit Transactions button.**

   This credits your transaction and displays a confirmation message.

Manually crediting a transaction

1. **Click on the Manual button.**

   This displays the Manual Credit page.

2. **Fill in the fields of the form.**

   The fields include credit card type, number and expiration, the cardholder's name, the amount, and a brief description of the credit.

3. **Click on the Perform Credit button.**

Viewing previous transactions by batch

1. **Click on the Previous button.**

   This page displays a list of transaction batches. Clicking on the batch ID displays the list of transactions in that batch.

Settling the current batch

1. **Click on the Settle button.**

   The **Settle Batch** button settles the current batch, which includes all transactions that are captured or capturing and credited or crediting. The current batch does not include authorized transactions.

The **Autocapture** button captures outstanding authorizations and settles the current batch of captured transactions.

Searching for a transaction

The LPAdmin application provides a query page you can use to search for a transaction. To search for a transaction:

1. **Click on the Query button.**

   If you are searching for a single transaction and you know the transaction's ID number, you may enter the ID and run the query. If you don't know the transaction ID or are searching for multiple transactions, there are additional fields to fill in.

2. **Enter the information in the fields provided.**
3. **Click on the Run Query button.**

   The query returns the transactions that matched *all* your parameters.

Chapter

5

Reviewing the application code

The Starter Application Set code provides you with code that performs all the basic credit card processing functions. As with LPStart, your application should call the functions defined in the .js libraries to perform its basic credit card processing tasks. The next section describes these functions.

If you wish to modify the Starter Application Set, see Chapter 6, “Modifying the Starter Application Set”.

Although this chapter also delves into the specifics of the .js libraries, since the library code is generic to any LivePayment application, you do not need to modify any portion of the libraries for your specific needs. The material in this chapter is intended to help you understand how to use the LivePayment objects in your code.

This chapter presents the following topics:

- The credit card processing functions
- The LiveWire objects
- The LivePayment objects
- Authorizing a purchase
- Capturing a transaction

- Crediting a transaction
- Settling a batch

The credit card processing functions

The following table describes the library functions that perform credit card processing tasks for the starter application. You should call one of these functions when you need to perform a credit card processing function.

Function	Description	File
GetCurrentBatch	Get the current batch of transactions	common.js
Authorize	Authorize a transaction	authorize.js
Capture	Capture an authorized transaction	capture.js
Credit	Credit a transaction	credit.js
Cancel	Cancel a transaction	cancel.js
SettleBatch	Settle the current batch of transactions	settlebatch.js

The credit card processing functions are described in depth in Chapter 7, “Netscape LivePayment Starter Application Set Reference”.

The LiveWire objects

This section reviews the LiveWire objects used in the .js libraries. You can find complete information about these objects and their usage in LiveWire applications in the *LiveWire Developer’s Guide*. If you are familiar with the LiveWire objects, skip to *The LivePayment objects* later in this chapter.

The database object

A LiveWire application uses the **Database** object to communicate with your database. Each application can have only one **database** object. You do not need to create the **database** object—it is created for you when you connect to the database. You connect to the database with the **connect** method.

```
database.connect
("server","server_name","user_name","password","database_name")
```

The execute method

Once connected, the application executes SQL queries on the database with the **execute** method. For example, the following call sets the status of purchases with id=authid in table **LP_PURCHASE** to CANCELLED.

```
database.execute("update LP_PURCHASE set status = \'CANCELLED\' where id
= " + authid)
```

The cursor method

The result of a SQL SELECT statement is stored in a **cursor** object. To create a **cursor** object, the application invokes the **database** object's **cursor** method. The syntax for a **cursor** method call is:

```
cursorName = database.cursor("sqlStatement",[updateable])
```

For example, the following call selects the rows from table **LP_PURCHASE** with id=transid and stores them in the *my_cursor* object.

```
my_cursor = database.cursor("select * from LP_PURCHASE where id='" +
transid + "'");
```

The application advances through the rows in the *my_cursor* object with the **cursor.next** method. For example, the following code displays rows (transactions) from a cursor in tabular form.

```
while (my_cursor.next())
{
     </SERVER>

     <TR>
     <TD><SERVER>write("<a href=\"viewtrans.html?type=" + request.type
+ "&batchid=" + my_cursor.id + "\">" + my_cursor.id + "</a>");</
SERVER></TD>
     <TD><SERVER>write(my_cursor.batchNumber);</SERVER></TD>
```

```
     <TD><SERVER>write(my_cursor.merchantReference);</SERVER></TD>
     <TD><SERVER>write(my_cursor.totalSalesAmount);</SERVER></TD>
     <TD><SERVER>write(my_cursor.totalCreditAmount);</SERVER></TD>
     <TD><SERVER>write(my_cursor.salesCount);</SERVER></TD>
     <TD><SERVER>write(my_cursor.creditCount);</SERVER></TD>
     <TD><SERVER>write(my_cursor.status);</SERVER></TD>
     </TR>
     <SERVER>
}
my_cursor.close()
```

The project object

The **project** object maintains global data for an entire application, such as the currency and acquirer. The **project** object allows multiple clients accessing a single application to share data.

The request object

The **request** object is created in response to a form submission. Each property of the **request** object corresponds to a form input element of the same name. For example, the following HTML code corresponds to the **request.guess** property.

```
<FORM METHOD="post" ACTION="hangman.html">
<P>
What is your guess?
<INPUT TYPE="text" NAME="guess" SIZE=1>
```

The LivePayment objects

This section provides brief descriptions of the LivePayment objects. These objects are described in depth in Chapter 8, “Using the LivePayment objects”, and Chapter 9, “LivePayment object reference”. If you are familiar with the LivePayment objects, skip to “Authorizing a purchase” on page 88.

The Merchant object

The **Merchant** object represents a merchant doing commerce on the Internet. It is used to store the information that identifies the merchant to the acquirer, such as the merchant number (provided by the acquirer) and the merchant name.

The Processor object

The **Processor** object is how LivePayment represents the bank card acquirer. The acquirer handles payment transactions from the merchant. The **Processor** object takes care of communicating with the acquirer through the gateway. The **Processor** has methods for authorizing, capturing, crediting, and settling transactions.

The Slip object

The **Slip** object is analogous to the paper slip used to confirm a charge to your credit card. It contains credit card information and order information. Unlike a paper slip, the **Slip** object encodes the credit card information for security. The encoded information is decoded by the acquirer to approve and collect the payment for the merchant.

The Batch object

The **Batch** object is a collection of transactions that have not yet been settled. The **Batch** object contains information about each of those transactions. Batch objects are identified by a batch ID.

The Terminal object

The *Terminal* object is analogous to a supermarket checkout terminal. It is where the card processing takes place. Each terminal is assigned a number by the acquirer. Just as a supermarket can have multiple checkout lanes, each with a separate terminal, your application may have multiple terminals.

Authorizing a purchase

The system authorizes a purchase by generating the **Slip** object, encoding the slip, and authorizing the credit card charge. The code for authorizing a purchase resides in the authorize.js file.

Generating a slip

By generating a slip, the application encodes the information relevant to a transaction in a format that can be transmitted to the acquirer bank for authorization.

The application generates a slip by creating a **Slip** object and populating its properties with the values garnered from the purchase form. The call that creates the **Slip** object is in the **GenerateSlip** function in file *common.js*:

```
slip = new Slip(slipData.cardNumber, slipData.cardExpDate, amount,
currency);
```

The call takes the credit card number, expiration date, amount, and currency and returns a new **Slip** object.

After the **Slip** object is created, additional values from the transaction are entered into the object's properties, including an order description.

```
slip.billingStreet = slipData.Address;
slip.billingZip = slipData.Zip;
slip.cardType = slipData.CCType;

slip.appendOrderDesc(orderDesc);
```

Encoding a slip

To encode a slip, the application puts the slip information into a DER (*Distinguished Encoding Rules*) encoded string. Although the application can recreate the slip from the string, it cannot access the card number once it is encoded. The **GenerateSlip** function encodes the slip, and then recreates it, thus protecting the credit card information.

The following call from function **GenerateSlip** encodes the **Slip** object, then extracts the resulting DER string:

```
if (!slip.encode(processor))
{
     PrintFmtError("Failed to encode slip",
          slip.getStatusMessage());
}
asciiDER = slip.getDER();
```

Next, **GenerateSlip** creates a **cursor** object, saves the DER string to the **cursor** object, and calls the LiveWire **database** object to save the encoded string to the database in the **LP_SLIP** table:

```
var nextSlipID = GetNextSlipID();
if ((error = database.beginTransaction()))
     PrintError("Could not begin a transaction in authorize().",
          "Error code " + error + " was returned.",
          "Please contact your system administrator.");

cursor = database.cursor("select * from LP_SLIP", true);
cursor.ID = nextSlipID;
cursor.slipDER = asciiDER;
if ((error = cursor.insertRow("LP_SLIP")))
{
     cursor.close();
     database.rollbackTransaction();
     PrintError("Failed to insert slip into database, error ", error);
}
cursor.close();

database.commitTransaction();
```

Having saved the encoded slip, **GenerateSlip** now recreates the slip from the encoded one. The new slip, slip2, contains information in encoded form:

```
slip2 = new Slip(asciiDER);
if (slip2.bad())
{
```

```
     PrintFmtError("Failed to construct slip object from DER.",
          slip2.getStatusMessage());
}
     slip2.initMerchantOrderDesc(amount, currency);
     slip2.appendMerchantOrderDesc(orderDescription);
     slipID = nextSlipID;// set Global var slipID
     return (slip2);
}
```

Authorizing a transaction

To authorize a transaction, the application creates a **PayEvent** object, and makes a call to the **authorize** method of the LivePayment **processor** object. This is how the **Authorize** function does it.

```
if ((authResult = processor.authorize(terminal, merchant, payevent,
     slip)))
{
     avsFail = CheckAVS(payevent);
     // Save transaction info to the database
     SavePayevent(payevent, slip, avsFail, authResult,
          authData.cardHolderName);
     if (avsFail)
     {
          PrintAVSError(avsFail);
     }
     else
     {
          returnval  = true; // else, no errors so return TRUE
     }
}
else
{
     // Save transaction info to the database
     SavePayevent(payevent, slip, false, authResult,
          authData.cardHolderName);
     PrintFmtError("Credit Card Authorization Failed.",
     processor.getStatusMessage());
}
```

In this code sample, **CheckAVS** calls the Address Verification Service code to ensure that the address ans zip code furnished with the purchase matches the cardholder's address and zip code.

Capturing a transaction

The authorized amount must be captured, or charged to the purchaser's credit card. This step usually takes place when the goods are to be shipped.

The LivePayment code for capturing a transaction is in capture.js. The following is the main code for capture.js:

```
// construct payevent object
//
if (merchantReference == null)
merchantReference = slip.merchantReference;

payevent = new PayEvent(merchantReference);
payevent.amount = purchaseCursor.amount;
payevent.authCode = purchaseCursor.authCode;
payevent.paySvcData = purchaseCursor.paySvcData;
payevent.avsResp = purchaseCursor.avsResp;
     payevent.eventID = eventID;

//
// Perform the capture
//
if (processor.capture(terminal, merchant, payevent, slip, batch))
{
     purchaseCursor.eventID = eventID;
     purchaseCursor.eventTime = processor.eventTime;
     purchaseCursor.status = "CAPTURED";
     if ((error = purchaseCursor.updateRow("LP_PURCHASE")))
     {
          purchaseCursor.close();
          database.rollbackTransaction();
          PrintError("Failed to update LP_PURCHASE in database,
          error",error);
     }
}
else
{
     purchaseCursor.close();
     database.rollbackTransaction();
     PrintFmtError("Failed to capture.",
     processor.getStatusMessage());
}
```

The **Capture** function searches through the database for capturing transactions. When it finds one, it recreates the **Slip** object from its DER encoded string stored in the **LP_SLIP** table, creates a new **PayEvent** object, and makes a call to the capture method of the **Processor** object to execute the capture. After the capture, the transaction is changed to captured status.

Crediting a transaction

Think of a credit as the reverse of a capture. To credit a transaction, the system must retrieve the transaction from that database, reconstruct the slip, and communicate the credit to the acquirer bank.

The **Credit** function in credit.js ensures that the current batch is valid and not busy with a settle, then calls the **DoCredit** function.

The **DoCredit** function searches the database by purchase ID, selecting transactions that are captured or crediting. When it finds the transaction to credit, it creates a new pay event with crediting status, reconstructs the slip, and performs the credit with the following call:

```
processor.credit(terminal, merchant, payevent, slip, batch)
```

If the credit call is successful, **DoCredit** calls **SaveCreditEvent** to record the credit event in the database for reference.

Settling a batch

The settling of the current batch is done in the settlebatch.js library. After obtaining the current batch, the system executes the following code.

```
preparetoSettle(batch, merchant, terminal, processor);
SettleBatch(batch, merchant, terminal, processor);
```

The call to **PrepareToSettle** checks that there are no incomplete transactions in the current batch and changes the status of the transactions to settling.

After making this check, the application can settle the batch. **SettleBatch** settles the batch by the following algorithm:

1. Add up the amount and number of purchases (debits) to settle.

2. Add up the amount and number of credits to settle.
3. Insert the sales and credit counts and totals into the batch object.
4. Call **processor.settleBatch** to settle the batch.
5. Update the appropriate row in the **LP_BATCH** database table with the transaction information.

Chapter

6

Modifying the Starter Application Set

In modifying the Starter Application Set, we recommend that you change only the LPStart HTML and .js files. By modifying other .js libraries, you could cause your application to behave incorrectly. However, if you do decide to modify the .js files, *make a backup of the original files* before making any modifications.

This chapter presents the following topics:

- Preparing to modify LPStart
- Modifying the Starter Application Set

Preparing to modify LPStart

Before modifying LPStart, you need to understand the following facts and assumptions:

- The Starter Application Set assumes a single merchant, and a single item per purchase.
- You will need to modify the Starter Application Set when you want to integrate your specific products and web pages with the payment functionality provided with Netscape LivePayment.

  For example, you'll need to modify the LPStart application if you wish to dynamically generate product information or integrate a shopping basket into your site.
- You are free to modify the look and feel of the LPAdmin administration application interface.
- When deploying your application, your modified LPStart must be run under the same LiveWire Application Manager as your LPAdmin in order for them to share similar data structures.
- Read the comments in the HTML files. They contain information to help you in building your own pages. In addition, read Chapter 7, "Netscape LivePayment Starter Application Set Reference".
- Have the code available when reading the remainder of this chapter.

Modifying the Starter Application Set

This section describes how to modify the LPStart application.

Modifying LPStart

When modifying the LPStart HTML pages, be careful of dependencies between files. Some HTML pages call on other pages to process form data. For example, the products.html page makes a call to the purchase.html page when a product

is purchased. The call is in the form of an ACTION executed when the purchase form is submitted. If you want to maintain the purchase.html file, you should provide the same parameters to the purchase.html file.

The following are the steps for modifying LPStart:

1. **Copy the starter application files to a new directory**

2. **Change the database name**

   Use the livepay.sql file to create your database, modifying the following line as follows:

   ```
   create database LIVEPAYMENT with log;
   ```

3. **Indicate what credit cards you support**

   Before creating the database for your application, specify which credit cards you will accept. Enter the credit card types into the cardtype.unl file. To add or delete a card type, simply add or delete it from this file.

Note When adding or removing credit card types, make sure the last line of cardtype.unl ends with a carriage return.

To add or remove credit card types *after* you build the Starter Application Set, do so through your database's SQL access tools (dbaccess for Informix, sqlplus for Oracle). Credit card types are stored in the **LP_CARDTYPE** table. See Chapter 7 for a description of the structure of the **LP_CARDTYPE** table.

4. **Create the new database**

   On Unix systems, run the *livepay.csh* script to create the database.

   On Windows NT (and Unix), you can use the Informix dbaccess tool to create the database using the livepay.sql SQL creation script. Refer to the Video sample application in the *LiveWire Developer's Guide* for instructions on creating databases using dbaccess.

5. **Replace or delete the sample HTML pages.**

   home.html—Replace or delete this page when you are ready to begin integrating the Starter Application Set into your web site; this page is only used to demonstrate the functionality of the application. If you delete this page, use the Application Manager's modification option to change the default page to a different page.

6. **Replace products.html with your own page.**

   The following parameter is required for generating a purchase amount and order description:

   productID—A string containing a unique identifier for a particular product. For example, a unique product ID from an inventory database. This value must be passed/submitted to *purchase.html*.

   If this parameter is not passed, the amount and order descriptions (request.amount and request.orderDesc) default to:

   amount: 0

   orderDesc: “NoDescription”

   The productID may be passed in one of two ways to purchase.html:

   - As an HTML form element named **productID**. For example, create a new page, such as:

```
<HTML>
...
<BODY>
...
<FORM method="POST" action="purchase.html">
...
<!-- Unique product ID for a Netscape Lamp Shade from inventory
database-->
<input type=hidden name="productID" value="i059a">
...
<input type=Submit name="Submit" value="Purchase Lamp Shade">
</FORM>
...
</BODY>
</HTML>
```

   The above HTML form contains two hidden form elements, one representing an amount of $50.00 (5000 cents), and one representing a description of the product (“Lamp Shade”). These fields are then passed on to purchase.html when the Submit button is clicked.

   - As URL-encoded name-value pairs

```
redirect("purchase.html?productID=i059a");
```

   In the above example, the product ID is passed as a parameter in the query string of the URL. See the *LiveWire Developer's Guide* for details about passing parameters in this fashion.

7. **Modify GetItemProperties function in purchase.js:**

   GetItemProperties is called from auth.html and returns a new instance of an **ItemObject**. To obtain your product/order amount and description based upon the productID passed into auth.html, you must modify **GetItemProperties** to perform a lookup in your own code or products database to find the item properties.

   Replace the existing sample code in this function with your own code to perform a lookup, be it a simple "if else" structure for a small number of products, instructions to read from a text file, or to read from a database. After finding a particular item, you must assign the amount of the item, in cents, and a description to a new instance of an **ItemObject** as shown in the example code:

```
if (productID == "001")
     item = new ItemObject("1800", "Mozilla Coffee Mug");
```

   The amount is the first argument and must be a string representing the purchase amount of the item in cents. The orderDesc is the second argument and should be a brief phrase describing the product. To protect against the possibility of errors, if no product is found, the following code should be executed, as shown in the example code in purchase.js:

```
item = new ItemObject("0", "Invalid Item");
```

   The item returned by this function (called from auth.html) is checked for validity to help prevent invalid items from being selected, so the above statement should be executed exactly as shown above if no product is found.

Example: Suppose a merchant wishes to replace the example code with a table lookup scheme based on an inventory table, PRODUCTS. The PRODUCTS table contains several fields: ID, description, color, weight, price. This example assumes that "price" is already stored in the database in cents as a string value and "description" is a brief description of the product. The merchant might write the following code to replace the example code for creating a new "ItemObject" to hold the purchase amount and description:

```
function GetItemProperties( productID )
{
// Establish a cursor to select the product we're looking for
cursor = database.cursor("select  amount, description from PRODUCTS
where ID='" + productID + "'");

// Attempt to initialize the cursor
if (cursor.next())
     item = new ItemObject(cursor.price, cursor.description);
```

```
     // Return an "invalid" ItemObject if no item was found
     else
          item = new ItemObject("0", "Invalid Item");
     cursor.close();
     return item;
} // END FUNCTION GetItemProperties
```

8. Modify CalcOrderTotal function in purchase.js

Use the **CalcOrderTotal** function, called by *auth.html*, to calculate the final total of a customer purchase. This is where a merchant might calculate sales tax, shipping & handling, fees, etc. The result returned from this function must be the total of the purchase, as a string, in cents.

Example Suppose a merchant wants to calculate tax and add shipping & handling to all orders. The **CalcOrderTotal** function might be modified as follows:

```
function CalcOrderTotal( amount, shipping, taxRate )
{
     // amount is the subtotal in cents
     // shippingCents is amount of shipping to be added to the
     // total
     // taxRate is the taxation rate as a percentage (e.g.  0.0825)

     // find amount with tax
     amount = amount * (1 + taxRate);

     // add shipping costs to get total of purchase
     amount = amount + shippingCents;

     // return the total amount to be authorized to "auth.html"
     return amount;

} // END FUNCTION CalcOrderTotal
```

Notice that the example above shows two new parameters were added: shipping and taxRate. This function may be extended as necessary, as the example shows. Take care to reflect any changes to the function definition in auth.html where CalcOrderTotal is called.

9. Modify these HTML pages.

start.html—Modify the PROJECT object parameters in the "Configurable Parameters" section at the top of the page. All of the parameter values should be enclosed in quotes, and correspond directly with the parameters required for a LiveWire "database.connect(...)" statement.

Warning Do not change the names of any parameters, only their values.

project.dbType - your database type (for example, "INFORMIX")

project.dbServer - your database server name

project.dbUsername - your database login name

project.dbPassword - your database password

project.dbName - the name of your database. Change this to something other than "LIVEPAYMENT." This should have the same value for LPAdmin and LPStart.

You may also modify other parameters in this section:

```
project.acquirer = "FDC"
```

Only change this if your acquirer is someone other than First Data Corporation.

```
project.currency = "USD"
```

Only change this if the currency type for your site is other than United States Dollars

```
project.failOnAddr = "Y"
```

Keep value of "Y" if you want authorizations to fail on AVS Address mismatches. Only change to "N" if you want to ignore AVS Address mismatches.

```
project.failOnZip = "Y"
```

Keep value of "Y" if you want authorizations to fail on AVS Zip code mismatches. Only change to "N" if you want to ignore AVS Zip code mismatches. Default is "Y."

```
build -o ...
```

You may wish to rename the .web file created by this build script for your own application. To do so, in the first line of the build file, change the filename indicated after the **-o** option. For example, to change the name of the application to MyPaymentApp.web from the default LPStart.web, change the first line to read:

```
build -o MYPaymentApp.web error.html home.html …
```

If you added or deleted files for your application, you need to add or delete entries in the build file.

10. Build and install the new application.

Run the build script from the application directory to build the new .web file. Install the application in the LiveWire Application Manager with an appropriate application name and external library parameters. See the entry for LPStart for the correct library name.

Refer to the *LiveWire Developer's Guide* for information on installing LiveWire applications.

Chapter

7

Netscape LivePayment Starter Application Set Reference

This chapter contains reference information for the Starter Application Set.

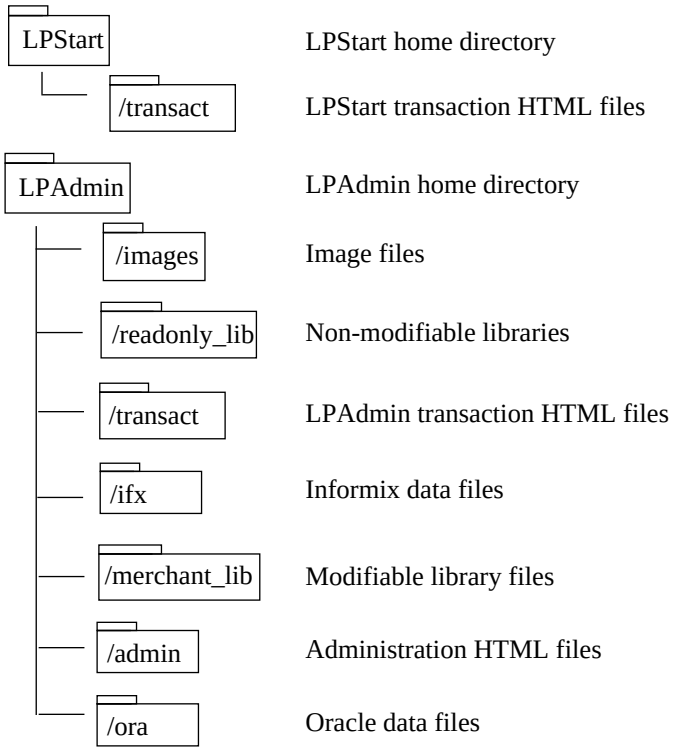
This chapter contains the following sections:

- Directory Structure
- LPStart HTML pages
- LPAdmin HTML pages
- LPAdmin Merchant Libraries
- Database schema
- Database tables
- Functions
- Batch states
- Transaction states

Directory Structure

The Starter Application Set directory structure is shown in Figure 7.1.

Figure 7.1 Starter Application Set directories



LPStart HTML pages

The following is a brief overview of the HTML files in the LPStart application.

Demonstration pages

home.html The main page for the LivePayment Starter Application Set. From here users can access the purchasing pages.

products.html A sample page provided for demonstration purposes. It contains a few sample products and shows how the purchasing and authorization process works in the Starter Application Set. This page would be modified or replaced with a merchant's product information or with functions that display data from a product description database.

purchase.html An HTML form where customers enter billing information such as credit card number, expiration date, address, and so on. May be modified by the merchant to include more information if desired.

Generic pages

error.html Prints a page indicating which particular error has occurred.

start.html Initializes object properties and establishes a database connection at application startup. It must be defined as the "initial page" in the LiveWire Application Manager.

Transaction pages

auth.html Form handler for the Customer Information form.

LPAdmin HTML pages

The following is a brief overview of the HTML files in the LPAdmin application.

Administration pages

The following are the administration pages in the LPAdmin application, found in the admin directory.

adminhelp.html	Displays help about administrative functions to administrators.
adminhome.html	Main menu for the administrative functions.
cancelauth.html	Used to cancel an authorized transaction.
changeauth.html	Gives the user a choice to capture or cancel a particular transaction.
changecred.html	Gives the user the ability to credit a particular transaction.
manualcredit.html	Used to perform a manual credit.
query.html	The main Query screen that allows users to select transaction criteria and run a query against the database.
querylist.html	A page that dynamically constructs a table that shows the results from a query. It allows users to click on a transaction ID to see more information.
search.html	The form handler for the query.html criteria selection form. Calls the functions that perform the query.
settle.html	Used to settle a batch, or auto-capture and then settle the batch.
transview.html	Dynamically constructed page that prints out information regarding a particular transaction in a two-column tabular format.
viewbatch.html	A dynamically constructed page that prints a list of batches currently stored in the database in HTML table format.
viewtrans.html	A dynamically constructed page that prints a list of transactions currently stored in the database in HTML table format. It accepts parameters as properties of the **Request** object to determine what kinds of transactions will be printed out (such as failed authorizations, captured transactions, and so on).

Generic pages

The following are the generic pages of the LPAdmin application, found in the lpadmin directory.

error.html	Form handler for the Customer Information form.
home.html	Welcome page
start.html	Initial application page. Designate this page as the initial page in the LiveWire Application Manager.

Transaction pages

The following are the transaction pages of the LPAdmin application, found in the transact subdirectory.

cancel.html	Form handler for the Customer Information form.
capture.html	Form handler for the **changeauth.html** form.
credit.html	Code for performing a credit transaction and form handler for **changecredit.html** form.
settlebatch.html	Form handler for the **settle.html** form.

LPAdmin Transaction Libraries

The following are the libraries that perform the credit card transactions for LPStart. These libraries are found in the readonly_lib subdirectory. They are read-only and should not be modified.

authorize.js	Functions relating to performing authorizations.
cancel.js	Functions relating to canceling authorized transactions.
capture.js	Functions relating to performing credit card captures.
common.js	Functions that are of a general nature. Contains some information retrieval functions, such default merchant reference generation functions.
credit.js	Functions relating to performing credits.
settlebatch.js	Functions relating to settling a batch of transactions.

LPAdmin Merchant Libraries

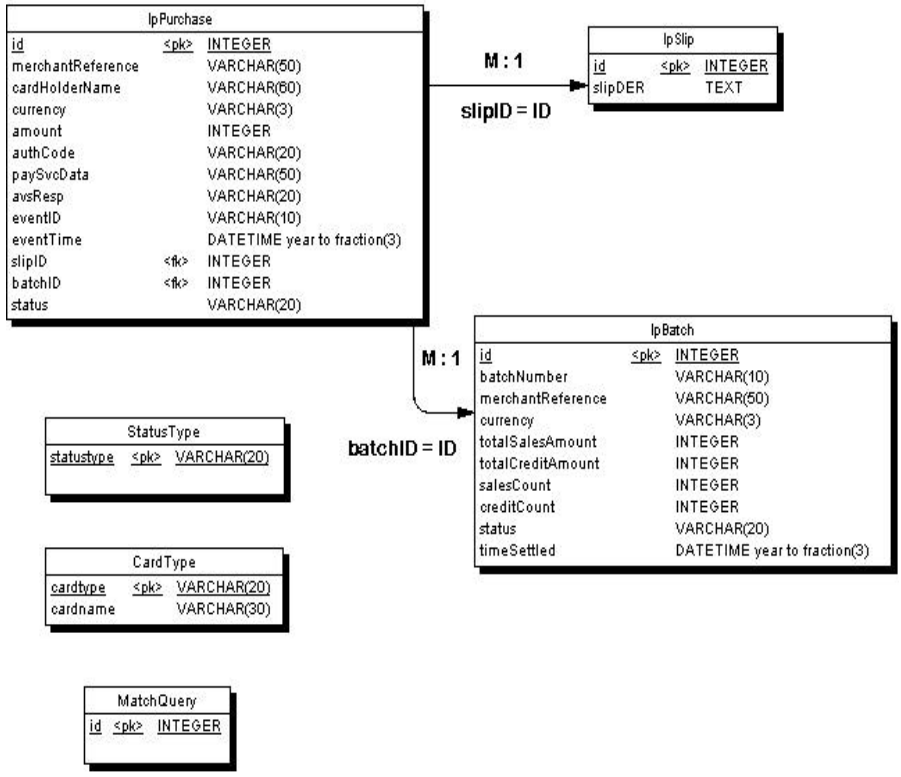
The following is a brief overview of the merchant (modifiable) libraries in the LivePayment Starter Application Set, found in the merchant_lib subdirectory. The functions available from those libraries are described on page 114.

config.js	Functions related to a site's configuration and validation of purchase form data.
dynamic.js	Functions that dynamically generate HTML output, such as dynamic generation of HTML drop-down list boxes.
error.js	Functions related to formatting and printing error messages in a standard format.
generateid.js	Functions that generate ID's for the database.
purchase.js	Functions used to purchase a product.
query.js	Functions that relate to running transaction queries against the database.
utility.js	Generic utility functions.
verify.js	Functions relating to verifying the correctness of data entered by the user, such as credit card number verification, zip code validation, and so on.

Database schema

The following figure illustrates the Starter Application Set's database schema.

Figure 7.2 Starter Application Set Database Schema



Warning You should not delete or rename the existing tables or fields, because the Starter Application Set code depends on them. Also, you should not change any datatypes, field lengths, or integrity constraints for existing fields.

You may add fields to existing tables, but at your own risk. You may add new tables to the Starter Application Set database with no ill effects; this is the preferred method of extending the database.

Database tables

The Netscape LivePayment Starter Application Set database structure consists of tables used to store the transaction data and internal Starter Application Set data required to track transactions. Many of the fields are used to contain property values of various Netscape LivePayment objects. Specific details of these properties can be found in Chapter 8, “Using the LivePayment objects”.

LP_CARDTYPE table

Description A lookup table containing credit card types that are accepted by the LivePayment Starter Application Set. Not all card types stored in this table may be used by the merchant.

Fields The **LP_CARDTYPE** table contains the following fields:

cardtype	Stores a string that the application uses to identify credit card types.
cardname	Stores a string that is formatted for display purposes. While the application uses the representation in the **cardtype** field, this field contains a label for the card type that can be displayed to users and which has a more pleasant look.

LP_BATCH table

Description Contains a record for each batch used by the Starter Application.

Fields The **LP_BATCH** table contains the following fields:

id	Contains a unique ID number assigned by the Starter Application Set. This number equates with the **batchID** field of the **LP_PURCHASE** table.
batchNumber	Contains the batch number assigned by the acquirer. **batchNumber** is a property of the **Batch** object. There is no correspondence between the **batchNumber** and the **id** field above.
merchantReference	Contains batch-related information provided by the merchant for tracking purposes. Note that there are 50 characters available for this field, but acquirers may limit this field to fewer characters. Some acquirers may limit data to only numeric characters as well. For example, FDC limits data to 10 numeric characters. This field is different from the **merchantReference** field in the **LP_PURCHASE** table. **merchantReference** is a property of the **Batch** object.
currency	Contains a three character string designating the currency of the transactions in the batch.
totalSalesAmount	Contains the total dollar amount of all credit card captures in the batch. **totalSalesAmount** is a property of the **Batch** object.
totalCreditAmount	Contains the total dollar amount of all refunds in the batch. **totalCreditAmount** is a property of the **Batch** object.
salesCount	Contains the total number of captures in the batch. **salesCount** is a property of the **Batch** object.
creditCount	Contains the total number of refunds in the batch. **creditCount** is a property of the **Batch** object.
status	Contains the current status of the batch (OPEN, SETTLING, or SETTLED).
timesettled	Contains the time and date the batch was settled.

LP_PURCHASE table

Description Keeps track of each transaction as it occurs. A new record is generated for each transaction undertaken by the LivePayment Starter Application Set.

Fields The **LP_PURCHASE** table contains the following fields:

id	A unique, sequential field designator assigned by the Starter Application Set for each record in the table.
merchantReference	Reserved for the merchants' use. The field can be used for an indicator the merchant might wish to include with the transaction, such as an invoice number. **merchantReference** is a property of the **PayEvent** object. Note that there are 50 characters available for this field, but acquirers may limit it to fewer characters. Some acquirers may limit data to only numeric characters as well. For example, FDC limits data to 10 numeric characters. This field is different from the **merchantReference** field in the **LP_BATCH** table.
cardHolderName	Contains the name of the person whose name appears on the credit card used in the transaction.
currency	Contains a three character string designating the currency of the purchase.
amount	Contains the dollar amount for which the transaction was made. **amount** is a property of the **PayEvent** object.
authCode	Contains the authorization code returned from the acquirer who authorizes the transaction to take place. **authCode** is a property of the **PayEvent** object.
paySvcData	Contains the payment service data provided by the acquirer. **paySvcData** is a property of the **PayEvent** object.
avsResp	Contains the results of the address verification query to the acquirer. **avsResp** is a property of the **PayEvent** object.
eventID	Contains an ID used for capturing or crediting. **eventID** is a property of the **PayEvent** object.
eventTime	Contains the time the authorize, capture, or credit is approved by the acquirer. **eventTime** is a property of the **PayEvent** object.
slipID	Contains a number used to link the **LP_PURCHASE** table with the **LP_SLIP** table.
batchID	Contains a number used to link the **LP_PURCHASE** table with the **LP_BATCH** table.
status	Contains the current status of the transaction. The possible transaction states that may appear in this field are stored in the **StatusType** lookup table.

LP_SLIP table

Description Contains a record for each slip generated during a transaction.

Fields The **LP_SLIP** table contains the following fields:

id	Contains a unique ID number which equates with the **slipID** field of the **LP_PURCHASE** table.
slipDER	Contains the ASCII slip DER returned by the **getDER** method of the **Slip** object. See the description of the **Slip** object for information on what data can be obtained from the stored slip DER data.

LP_MATCHQUERY table

Description Used to store results from administrative queries run from the Query tool. It is cleared every time a new query is run.

Fields The **LP_MATCHQUERY** table contains the following field:

id	A unique integer that corresponds with the ID of a transaction.

LP_STATUSTYPE table

Description A lookup table containing all available transaction and batch states. See “Transaction states” on page 115 and “Batch states” on page 115 for a list of all possible states which are stored in this table.

Fields The **StatusType** table contains the following field:

statustype	Contains a static list of all possible status types (OPEN, CREDITED, and so on.). This field is not currently modified by any of the processes in the Starter Application Set.

Functions

The following is a brief overview of the functions available in the LivePayment Starter Application Set. These functions are available for the merchant's use and/or modification should it be required when integrating the Starter Application Set with a merchant's web site. Refer to the appropriate source code file for specific details pertaining to a given function.

Authorize

Description Used to authorize a new payment transaction. If the authorization fails, the function does not return. Instead, it calls one of the error functions, **PrintFmtError** or **PrintError**.

Library authorize.js

Parameters *authData* is an object that contains information about the customer, credit card, and order description. The specific properties of **authData** that are used in this function are:

amount - the amount of the transaction

currency - the currency type (e.g. USD)

orderDesc - a description of the order

cardHolderName - Full name as it appears on the card

cardNumber - credit card number

cardExpDate - expiration date

cardType - credit card type

address - address of card holder

zip - zipcode of card holder

merchantReference - a reference number that the merchant may assign to the transaction

Returns TRUE, if the operation succeeds. The function does not return if the authorization fails, but rather calls one of the error functions, **PrintFmtError** or **PrintError**.

AuthToCapturing

Description Changes the state of all authorized transactions to CAPTURING. Call the **GetCurrentBatch** function prior to calling **AuthToCapturing** in order to select the current open batch.

Library capture.js

Parameters *batch* is the **Batch** object associated with the transaction to be captured.

purchaseid is the unique id of the transaction to capture.

all is a boolean flag that tells whether or not to capture ALL AUTHORIZED transactions. If TRUE, all AUTHORIZED transactions will be changed to CAPTURING. If FALSE, then only the transaction indicated by *purchaseid* has its status changed to CAPTURING.

Returns None.

CalcOrderTotal

Description Calculate the total amount of the purchase.

Library purchase.js

Parameters *amount* is a sub-total for a purchase.

Returns The total amount of the purchase, including any modifiers such as shipping, state or local taxes, processing fees, etc.

CalcScore

Description Matches all of the user selected criteria values from the Query screen with the stored values in the database for each transaction stored in the **LP_PURCHASE** table. It returns a score, which represents the number of criteria values that exactly matched corresponding values in the database for a particular transaction.

Library query.js

Parameters *scursor* is a cursor pointing to a particular record in the **LP_PURCHASE** table.

Returns A numeric value representing how many of the parameters on the HTML query form matched values of the corresponding fields in the database.

Cancel

Description Changes the status of an authorized transaction from AUTHORIZED to CANCELLED in the database. Note that this function does not cancel the authorization with the acquirer. In order to cancel a transaction that has already been submitted to an acquirer, use Credit.

Library cancel.js

Parameters *authid* is the ID of the transaction that will be cancelled.

Returns TRUE if the operation is successful.

Capture

Description Looks for transactions which have a status of AUTHORIZED or CAPTURING and attempts to capture those items. Upon success, it changes the transaction status to CAPTURED. See the library file for details on this function.

Library capture.js

Parameters *id* is the ID of the purchase as it relates to the **LP_PURCHASE** table.

batch is the **Batch** object associated with the transaction to be captured.

merchant is the **Merchant** object, previously created.

terminal is the **Terminal** object, previously created.

processor is the **Processor** object, previously created.

merchantReference is the merchant reference string for the capture.

all is a boolean variable. If TRUE, capture all transactions in the CAPTURING state. Otherwise, capture only the transaction indicated by "id" parameter.

Returns TRUE if the capture was successful; FALSE if it was not.

CentsToDollarStr

Description Converts a monetary value expressed in cents into a monetary value expressed in the dollars.cents format by shifting the decimal point two positions to the left.

Library authorize.js

Parameters *cents* is a string representing a value in cents.

Returns A string representing the input value in dollars.

CheckAVS

Description Determines if the Address Verification Service (AVS) checks on the address and zip code pass or fail.

Library authorize.js

Parameters *payevent* is a *PayEvent* object that contains AVS data returned by the acquirer after the authorization.

Returns An integer indicating which part of the AVS check failed: address or zip. Returns a 0 for no failures, 1 for address fail, 2 for zip fail.

Confirm

Description Accepts a boolean argument and a string argument. The boolean argument is used to determine whether or not to print out a message confirming success for an arbitrary operation, or failure for that operation. The string argument is used to print out what type of operation is to be confirmed as successful or not.

Library dynamic.js

Parameters *success* is a boolean variable indicating whether or not to print out a message of success or failure of an operation.

type is a string to indicate what type of operation succeeded or failed (such as CREDIT).

Returns None.

CountActiveParams

Description Counts the number of criteria on the Query page for which the user has filled in values. It returns the total number of non-blank criteria fields. See the library file for details on this function.

Library query.js

Parameters None.

Returns The number of items in the query form which contain values to be matched.

CreateCreditEvent

Description Creates a new record in the database and sets its status to CREDITING. See the library file for details on this function.

Library credit.js

Parameters *batch* is a batch object associated with the current batch.

purchaseid is the ID of the CAPTURED transaction that is to be credited.

Returns A *PayEvent* object associated with the credit transaction.

CreateManualCreditEvent

Description Creates a new **PayEvent** object, initializes some of its properties and sets its status to CREDITING.

Library credit.js

Parameters *batch* is a batch object associated with the current batch.

merchantReference is a transaction reference number created by the merchant.

creditSlipID is the slipID used when inserting the transaction record.

creditCardHolderName is the name of the card holder.

credtAmount is the dollar amount of the credit transaction.

Returns The **PayEvent** object.

CreatePayEvent

Description Creates a new record in the **LP_PURCHASE** table and sets its status to AUTHORIZING.

Library authorize.js

Parameters *amount* is the amount of the authorization.

Returns A *PayEvent* object initialized with the merchantReference, amount, and ID.

Credit

Description Creates a new record in the **LP_PURCHASE** table and sets its status to CREDITING. This function calls the **DoCredit** function upon successful completion.

Library credit.js

Parameters *batch* is the batch object associated with the current batch.

All other parameters are simply passed through to the **DoCredit** function, which is called from within this function.

Returns TRUE if the batch is in the OPENED status and the credit may take place; FALSE otherwise.

DoCredit

Description Performs the actual credit operation that refunds the previously captured amount to the customer's account.

Library credit.js

Parameters *batch* is the **Batch** object associated with the current batch.

purchaseid is the ID of the credit transaction which is currently in the CREDITING state.

merchant,

terminal,

processor are objects required for the credit transaction to take place (previously created).

Returns TRUE if the credit is successful; FALSE otherwise.

DollarStrToCents

Description Converts string dollar values to cents.

Library utility.js

Parameters *dollarstr* is a string representing a dollar amount.

Returns A numeric value representing the input string converted from dollars to cents.

DynSelect

Description Dynamically generates an HTML selection list box based on data stored in a lookup table in the database.

Library common.js

Parameters *listname* is the name to be given to the HTML select box

size is the size of the select box, where a 1 indicates a dropdown box, and greater than one indicates a scrolling list box.

multiple is a boolean variable indicating if multiple selections are allowed (TRUE indicates multiple selections are allowed).

handlerText is a string representing a client-side JavaScript event handler for the select box.

selectValue is a string representing the item to be selected by default upon the lists creation. "" indicates no selection by default.

blankline is a boolean value. If it is TRUE, a blank line will appear in the select box; if it is FALSE, no blank line will appear.

Returns None.

EmitFooter

Description Writes a standard closing HTML block to the screen.

Library dynamic.js

Parameters None.

Returns None.

EmitHeader

Description Writes a standard HTML header to the screen. It accepts a string as input for the HTML <TITLE> block.

Library dynamic.js

Parameters *title* is a string that contains the title of the page.

Returns None.

FailOnAVS

Description Returns a boolean value indicating whether an authorization should fail based on the AVS checks that were provided. This function determines if the authorization will fail by calling **project.failOnAddr** and **project.failOnZip**. See the library file for more on this function.

Library config.js

Parameters *avsFail* is an input parameter representing the numeric code resulting from CheckAVS indicating what kind of AVS failure occurred, if any. If *avsFail* is 1, it indicates an address failure, if *avsFail* is 2, it indicates a zip code failure.

Returns TRUE if the transaction should fail because of the AVS failure, or FALSE if it should succeed.

GenerateMerchantReference

Description Create a **merchantReference** number based on the purchase ID. This creates a default merchant **Reference** if the merchant does not specify one.

Library common.js

Parameters None.

Returns The **merchantReference** number.

GenerateSlip

Description Generates an encrypted slip for the current transaction and stores it in the **LP_SLIP** table of the database.

Library authorize.js

Parameters *request* is a copy of the **Request** object that contains the following properties

used in this procedure:

amount - the amount of the transaction.

currency - the currency type (e.g. USD).

orderDesc - a description of the order.

cardNumber - a credit card number.

cardExpDate - expiration date.

cardType - credit card type.

address - address of card holder.

zip - zipcode of card holder.

merchantReference - a reference number that the merchant may assign to the transaction

Returns The constructed **Slip** object.

GetCardType

Description Given a particular slip ID, this function returns the **cardType** indicated by that slip. If no slip matches the slip ID, this function returns an empty string ("").

Library common.js

Parameters *slipid* is the ID of the slip associated with the transaction in question.

Returns Given a particular slip ID, this function returns the **cardType** indicated by that slip. If no slip matches the **slipID**, this function returns an empty string ("").

GetCurrentBatch

Description Returns the current batch number if a batch exists with the status OPEN or SETTLING. If no batch exists with the status of OPEN or SETTLING, then a new batch number is obtained from the acquirer and its status is set to OPEN. See the library file for details on this function.

Library common.js

Parameters *merchant*, *terminal*, *processor*, are objects required for obtaining a new batch number from acquirer (previously created).

Returns The current batch number according to the acquirer.

GetItemProperties

Description Returns an **ItemObject** containing the cost and description of the product.

Library purchase.js

Parameters *productID* is a string representing a unique identifier for a product for sale on the web site.

Returns An **ItemObject** containing the cost of the object in a whole integer (e.g. cents) and a brief description of the product.

GetNextBatchID

Description Gets the next sequential batch ID number by incrementing the current batch ID number, stored as a property of the **Server** object, by one.

Library generateid.js

Parameters None.

Returns The next available, unique ID for use in the **Server** table when a new batch record is created (when a new batch is opened).

GetNextEventID

Description Gets the next sequential event ID number by incrementing the current event ID number, stored as a property of the **Server** object, by one.

Library generateid.js

Parameters None.

Returns The next available, unique ID for use in the **LP_PURCHASE** table when a new event is created in a particular batch.

GetNextPurchaseID

Description Gets the next sequential purchase ID number by incrementing the current purchase ID number, stored as a property of the **Server** object, by one.

Library generateid.js

Parameters None.

Returns The next available unique id for the **LP_PURCHASE** table.

GetNextSlipID

Description Gets the next sequential slip ID number by incrementing the current slip ID number, stored as a property of the **Server** object, by one.

Library generateid.js

Parameters None.

Returns The next available unique id for the **LP_SLIP** table.

GetTitleString

Description Creates dynamic titles for pages that are used for multiple views of data. Returns an HTML title string appropriate for the type of data being displayed on a page.

Library dynamic.js

Parameters *viewtype* is a type of list requested can be one of 5 types:

"auth" - authorized transactions to capture or cancel

"credit" - captured transactions to credit

"view" - view only, no hyperlinks on transaction id

"current" - transactions in current batch

"errors" - view failed transactions

batchid is the ID of the batch a transaction belongs to, if appropriate for the viewtype.

Returns A string of HTML that contains a title appropriate for the type of transaction list that will be displayed in **viewtrans.html**.

IsAmericanExpress, IsAmEx

Description Determines if a given credit card number is a valid number for American Express.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid American Express card number; FALSE, otherwise.

IsAnyCard

Description Determines if a credit card number is a valid number. It tests the number against all accepted card types (such as Visa, MasterCard, and so on) and calls **IsCC** to perform Luhn Mod-10 testing on the number. It returns a boolean value indicating success or failure.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is any valid credit card number for any of the accepted card types; FALSE, otherwise.

IsBlank

Description Used to test for an empty or null string. Not used for boolean or numeric variables. Returns TRUE if the string is empty or null; otherwise, it returns FALSE.

Library utility.js

Parameters *teststr* is the string to be tested to ensure that the string is null.

Returns TRUE, if the string is null or is an empty string, ""; otherwise FALSE.

IsCardMatch

Description Determine if a credit card number is valid.

Library verify.js

Parameters *cardType* is a string representing the credit card type.

cardNumber is a string representing a credit card number.

Returns TRUE if the credit card number is valid for the particular credit card type given in *cardType*.

IsCarteBlanche, IsCB

Description Determines if a given credit card number is a valid number for Carte Blanche.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid Carte Blanche card number; FALSE, otherwise.

IsCC

Description Determines if a credit card number passes the Luhn Mod-10 test.

Library verify.js

Parameters *st* is a string representing a credit card number.

Returns TRUE, if the credit card number passes the Luhn Mod-10 test; FALSE, otherwise.

IsDinersClub, IsDC, IsDiners

Description Determines if a given credit card number is a valid number for Diner's Club.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid Diner's Club card number; FALSE, otherwise.

IsDiscover

Description Determines if a given credit card number is a valid number for Discover.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid Discover card number; FALSE, otherwise.

IsEnRoute

Description Determines if a given credit card number is a valid number for en Route.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid en Route card number; FALSE, otherwise.

IsExistingPurchaseID

Description Determine if a purchase ID has an entry in the **LP_PURCHASE** table.

Library utility.js

Parameters *id* is the id of the transaction in question.

Returns TRUE if the ID exists, FALSE otherwise.

IsJCB

Description Determines if a given credit card number is a valid number for JCB.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid JCB card number; FALSE, otherwise.

IsMasterCard, IsMastercard, IsMC

Description Determines if a given credit card number is a valid number for MasterCard.

Library verify.js

Parameters *cc* is a string representing a credit card number.

Returns TRUE, if the credit card number is a valid MasterCard card number; FALSE, otherwise.

IsNum

Description Used to test for a numeric string variable. Returns

Library utility.js

Parameters *numstr* is a string that will be tested to ensure that each character is a digit.

Returns TRUE if all characters are from 0-9; otherwise, it returns FALSE.

IsValidDay

Description Validates a day of the month.

Library utility.js

Parameters *daystr* is a string representing a day of the month.

Returns TRUE, if the string represents a number from 1 to 31; otherwise, it returns FALSE.

IsValidPrice

Description Validates a string representing a price.

Library utility.js

Parameters *pricestr* is a string representing a price.

Returns TRUE, if it the value is a valid price, containing characters from 0-9 and not more than one (.) period for an optional decimal place. Returns TRUE if these criteria are met; otherwise, it returns FALSE.

IsValidPurchaseID

Description Validates a given purchase ID. Returns TRUE if the ID exists in the **LP_PURCHASE** table; otherwise, it returns FALSE.

Library common.js

Parameters *id* is the ID of the transaction in question.

Returns TRUE, if the ID is valid and exists in the **LP_PURCHASE** table; otherwise, it returns FALSE.

IsValidYear

Description Validates a string representing the year. Returns TRUE if the string is a 4-digit number; otherwise, it returns FALSE.

Library utility.js

Parameters *yearstr* is a string representing a 4-digit year.

Returns TRUE if the string represents a 4-digit number; otherwise, it returns FALSE.

IsValidZip

Description Determines if the zip code provided is a 5-digit string containing only numeric characters. It is used to verify zip codes provided in the customer information form.

Library config.js

Parameters *zipstr* is a string representing a 5-digit zip code.

Returns TRUE, if the string is 5-digits long; otherwise, it returns FALSE.

IsVISA, IsVisa

Description Determines if a given credit card number is a valid number for VISA.

Library verify.js

Parameters *cc* is a string representing a credit card number

Returns TRUE, if the credit card number is a valid VISA number; FALSE otherwise.

ItemObject

Description An object constructor intended to represent a typical item for sale on a web site, comprised of an amount, and a description. You may modify this constructor to hold additional information.

Library purchase.js

Parameters *amount* is the amount/cost of the item; it must be a whole integer (e.g. cents)

description is a short description of the item.

Returns None.

MakeValidMerchantReference

Description Trims down **merchantReference** string, if necessary, to be within the acceptable number of characters for the given acquirer. Returns the **merchantReference** string.

Library common.js

Parameters *acquirer* is a string representing the acquirer.

merchantReference is a string representing a **merchantReference** identifier to be trimmed, if necessary.

Returns A **merchantReference** string that is to be trimmed, if necessary, to the acceptable number of characters for the given acquirer.

MatchShipToBill

Description Maps billing properties of the **Request** object to the shipping properties in the case a user has checked a box on the form indicating that Shipping Information is the same as Billing Information.

Library config.js

Parameters Request is the LiveWire **Request** object resulting from a call from **purchase.html** and containing properties associated with the form elements contained within the form in **purchase.html**.

Returns Modified **Request** object properties for shipping information to match the corresponding billing information.

Note If the form element names from "purchase.html" referenced below are changed, this function must be altered to reflect the changes to those HTML form-element names, or mapping may cease to function correctly.

PrepareToSettle

Description Changes the batch status to SETTLING. It then calls the **Credit** and **Capture** functions to attempt to complete any pending credits or captures still in the CREDITING or CAPTURING state before settling the batch. This function should be called prior to calling **SettleBatch**, which actually settles the batch with the acquirer. See the library file for details on this function.

Library settlebatch.js

Parameters *batch* is the **Batch** object related to the batch being settled.

merchant,*terminal*,*processor* are objects, previously created, that are required to perform a **processor.settle** operation.

Returns None.

PrintAVSError

Description Prints an output message describing the reason for an AVS failure. Failure is either due to an address or zip code mismatch.

Library dynamic.js

Parameters *avsFail* is an integer indicating the failure/success status of the AVS data returned by the Acquirer after an authorization.

Returns None.

PrintBatchItem

Description A generic function called by the **PrintSettledBatch** function to format the screen output of a two-column batch report.

Library dynamic.js

Parameters *label* is the label for a row item in a table.

val is a string or number to be printed as a row item described by the label.

Returns None.

PrintError

Description Dynamically generates an HTML error page based on the parameters passed to the function. It is used to print out errors of a generic nature, not associated with a transaction failure.

Library error.js

Parameters *premsg* is a string indicating the first message to be printed. This first message is usually the "title" of the error.

info is the description of the error.

postmsg is usually a string containing a suggested course of action to resolve the error.

Returns None. Sets CLIENT object properties: **lpErrno**, **lpErrTitle**, **lpErrMsg** and **lpErrEnd**, then redirects to **error.html**.

PrintFmtError

Description Dynamically generates an HTML error page based on information received from the processor. It is intended for use when a processing error has occurred and information about that error must be returned to the user. For example, if a batch fails to settle because the totals do not match between the merchant and acquirer, this function is called to report the specific error that the **Processor** object returned.

Library error.js

Parameters *premsg* is a string indicating the first message to be printed. This first message is usually the "title" of the error.

fmtmsg is a string that describes the error. Used in this sample application to pass the status message from the processor method **getStatusMessage**.

Returns None. Sets CLIENT object properties: **lpErrno**, **lpErrTitle** and **lpErrMsg**, then redirects to **error.html**.

PrintReceipt

Description Used to output the data relevant to a given purchase to the screen by the **PrintReceiptItem** function. The output from this function may be printed from the customer's browser and used as a receipt. This function may be customized.

Library dynamic.js

Parameters *pay* is the **PayEvent** object associated with the purchase.

desc is a string describing the purchase.

Returns None.

PrintReceiptItem

Description A generic function called by the **PrintReceipt** function to format the screen output of a transaction receipt line item.

Library dynamic.js

Parameters *label* is the label for a row item in a table.

val is a string or number to be printed as a row item described by the label.

Returns None.

PrintSettledBatch

Description Used to output the data relevant to a given batch to the screen by the **PrintBatchItem** function.

Library dynamic.js

Parameters *batch* is a **Batch** object that contains information to be printed.

Returns None.

PrintTrans

Description Dynamically creates a screen output of data regarding a particular transaction.

Library dynamic.js

Parameters *cursor* is a cursor object that points to the transaction.

Returns None.

PrintTransItem

Description Called by **PrintTrans** and generates formatting for each line item passed to it.

Library dynamic.js

Parameters *label* is the label for a row item in a table.

val is a string or number to be printed as a row item described by the label.

Returns None.

Query

Description Accepts all of the criteria values the user entered on the Query screen and finds all of the transactions that exactly match all of the non-blank values entered. It determines a match by checking the score of the query for each transaction (see **CalcScore**) against the count of non-blank criteria values (see **CountActiveParams**). If the numbers are equal there is a match, and the ID of the matched transaction is stored in the **LP_MATCHQUERY** table. This function loops through every transaction in the **LP_PURCHASE** table.

Library query.js

Parameters None.

Returns None.

QueryCleanup

Description Deletes all records from the **LP_MATCHQUERY** table from the previous query to prepare for a new set of results.

Library query.js

Parameters None.

Returns None.

RemoveAlpha

Description Removes dashes, spaces, and other non-numeric characters from a credit card number.

Library utility.js

Parameters *cardNumber* is a credit card number.

Returns The credit card number with all dashes, spaces, and other non-numeric characters removed.

SaleObject

Description An object constructor used to contain and pass values associated with a purchase to the **Authorize** function or for a manual credit. It is intended to be called from a form handler, which formats user input before calling **Authorize** or **GenerateSlip**.

Library common.js

Parameters *amount* is the amount of the sale, as a non-zero, non-negative integer (e.g. must be a value in cents, for US currency).

currency is the currency for the transaction (3-letter code).

orderDesc is a description of the order

cardHolderName is the first and last name of the credit card holder, as it appears on the card.

cardType is the type of credit card (e.g. Visa, MasterCard, etc.).

cardNumber is a credit card number (must contain only digits 0-9, no spaces or dashes).

cardExpDate is the card expiration date (must be of the form YYYYMM).

address is the billing address for the card.

zip is the billing zip/postal code for the card.

merchantReference is a string for future reference.

Returns A new instance of the object.

SaveCreditEvent

Description Saves the processor information about a particular transaction to the **LP_PURCHASE** table and changes the transaction state to CREDITED. This function is to be called *after* the actual credit has been performed via communication with the acquirer across the Internet.

Library credit.js

Parameters *payevent* is a payevent object associated with the completed credit.

slip is a slip object associated with the purchase that was credited.

Returns None.

SavePayEvent

Description Saves the processor information about a particular transaction to the **LP_PURCHASE** table and changes the transaction state to AUTHORIZED.

Library authorize.js

Parameters *payevent* is the **PayEvent** object previously constructed that contains AVS data returned by the acquirer after the authorization.

slip is the previously constructed slip object associated with the authorization.

avsFail is the code indicating AVS failure or success.

authSucceed is the boolean value returned by the authorization indicating success or failure of the authorization.

Returns None.

SettleBatch

Description Settles the current batch with the acquirer and changes the transaction state to SETTLED.

Library settlebatch.js

Parameters *batch* is a **Batch** object related to the batch being settled.

merchant, *terminal*, and *processor* are objects, previously created, that are required to perform a **processor.settle** operation.

Returns None.

StateList

Description Returns a dynamically generated dropdown list box for selecting a state

Library dynamic.js

Parameters *listname* is the name of the dropdown list to be created.

site is the size of the dropdown list to be created.

multiple is TRUE or FALSE if the list should allow multiple selections.

handlerText is the optional JavaScript handler for the selected value.

selectValue is the default value to be selected.

Returns A dynamically generated dropdown list box for selecting state.

VerifyPurchaseData

Description Used to validate the user input provided in the Customer Information form (purchase.html). It verifies that required fields are not left blank, and where possible it validates the data input into the fields by calling such functions as **IsAnyCard** and **IsValidZip**.

Library config.js

Parameters *request* is a copy of the request object from the **purchase.html** input form.

Returns None.

Notes This function could be modified to validate additional fields from **purchase.html**.

ViewTransaction

Description Dynamically generates a listing of information regarding a given transaction. The information is presented in two-column tabular format on the screen.

Library dynamic.js

Parameters *transid* is an integer specifying the transaction from which information will be taken and output to the user.

Returns None.

Batch states

The following states exist for a batch. The current state for any given batch is stored in the **status** field of the **LP_BATCH** table:

OPENED	The batch is currently open. There will only be one batch in this state at any given time.
SETTLING	The batch is currently in the process of being settled with the acquirer. There will only be one batch in this state at any given time. No captures or credits may take place while a batch is in this state.
SETTLED	The batch has been properly settled with the acquirer.

Transaction states

The following states exist for a transaction. The current state for any given transaction is stored in the **status** field of the **LP_PURCHASE** table:

AUTHFAILED	The transaction was not authorized by the acquirer.
AUTHORIZED	The transaction has been authorized by the acquirer.
AUTHORIZING	The processor is currently attempting to authorize the transaction with the acquirer.
AVSFAILED	The address verification failed. For the default Starter Application Set configuration, any AVS failure on zip code or address fields equates to a failed authorization. This behavior may be modified in the code if desired.
CANCELLED	The authorization was canceled by the merchant.
CAPTURED	The authorized funds for the transaction have been successfully captured.
CAPTURING	The authorized funds for the transaction are in the process of being captured.

CREDITED	The previously credited funds for the transaction have been successfully refunded to the customer's account.
CREDITING	The previously captured funds for the transaction are in the process of being refunded to the customer's account.
SETTLED	The captured funds for the transaction have been collected
SETTLING	The captured funds for the transaction are in the process of being collected.

3

Creating a LivePayment application from the ground up

- **Using the LivePayment objects**
- **LivePayment object reference**

Chapter

8

Using the LivePayment objects

This chapter provides an overview and description of the LivePayment LiveWire capabilities. It includes a brief description of the LPAuthOnly sample application. This chapter assumes you are familiar with the basics of LiveWire and JavaScript. You should have already read the *LiveWire Developer's Guide*, which explains how to use LiveWire. You should have some programming experience with a programming language such as Pascal, C, or Visual Basic. Some background in object-oriented programming is recommended.

Before writing your LiveWire credit card processing application, you should read Chapter 3, "Payment application concepts", as well as this chapter.

This chapter contains the following sections:

- LivePayment objects overview
- Developing with LivePayment
- Using LivePayment objects to process payments
- Using the LPAuthOnly sample application

LivePayment objects overview

LivePayment contains objects for use with the LiveWire development environment. LiveWire enables you to create server-based applications similar to Common Gateway Interface (CGI) programs.

LiveWire's scripting language is called *JavaScript*. JavaScript is a compact, object-based scripting language for developing client and server Internet applications.

LivePayment provides you with an additional set of predefined JavaScript *object types* to use on the server to accept and process credit card transactions on a web site. An *object* is a construct with properties. *Properties* are JavaScript variables and can be other objects. Functions associated with an object are known as *methods*.

LivePayment contains the following objects:

- **Batch**
- **Merchant**
- **Processor**
- **PayEvent**
- **Slip**
- **Terminal**

For information on using these objects, see "Using LivePayment objects to process payments" on page 152.

Developing with LivePayment

In general, developing a LiveWire application using the LivePayment objects is very similar to developing any LiveWire application. This section contains instructions for using the LivePayment objects in LiveWire. It does not contain an explanation of how to develop a LiveWire application. For general instructions on developing an application in LiveWire, see the *LiveWire Developer's Guide*.

To construct an application using LivePayment's objects, follow these basic steps:

1. Install the Enterprise server, LiveWire, the database, and LivePayment. To install your database, follow the installation instructions that came with your database. To set up your database, plan out what information you want to store there and how to store it. For ideas on what tables and fields you might use, see the Starter Application Set.

2. Create the source files for the payment application in Navigator Gold or a text editor and save them on the server. When you create your application, you need to do the following:
   - Use the <SERVER> tags to embed your JavaScript in HTML.
   - Use **registerNativeFunction** in your application's initial page.
   - Use **registerLivePayment** to register the LivePayment objects on each page that calls them (if necessary).
   - Develop your payment application, using LivePayment's objects, properties, and methods. Refer to Chapter 3, "Payment application concepts" for more information. Also refer to the sample applications.
   - Use the LivePayment error status methods to check for errors in your application.

3. Build the application in LiveWire (or rebuild it, if it has already been built), using the site manager or the LiveWire compiler.

4. Install the application in LiveWire if it is not installed already. If it is already installed, restart it.

5. Run the application.

6. Creating an application is an iterative process. After creating, building, installing, and running the application, you will probably need to change the application, rebuild it, restart it, and run it again.

7. Run the application in loopback mode and test mode before using it in production mode. For more information, see "Changing LivePayment operating modes" on page 28.

8. Get your application certified by your acquirer before running live.

9. Run the application in production mode.

For information on installing the Netscape products and the database, see the documentation for those products. For information on building, installing, and running LiveWire applications, see the *LiveWire Developer's Guide*.

Embedding LivePayment JavaScript in HTML

When you embed your JavaScript code within HTML, all references to the objects, properties, and methods must be within the HTML tags <SERVER> and </SERVER>. These tags indicate that the objects can only be invoked and executed on the server side of LiveWire, as opposed to the client side. The tag <SERVER> precedes the JavaScript statements, and </SERVER> follows them. The tags enclose one or more JavaScript statements. For more information, see the LiveWire documentation.

Registering LivePayment objects in LiveWire

The LivePayment objects are not native LiveWire objects. There are two steps involved in registering the LivePayment objects:

1. On the project's initial page, register the function that registers LivePayment objects as native functions in LiveWire (**registerNativeFunction**).

2. Depending upon how you used **registerNativeFunction**, before the objects are called on a page you may need to register the LivePayment objects with LiveWire on that page (**registerLivePayment**).

First you need to register the **registerLivePayment** function and the shared library in which the function is located. You use **registerNativeFunction** on the project's initial page (the page that is loaded when the client first accesses the application). The following example shows registering the objects and the library:

```
<HTML>
<TITLE>Netscape LivePayment</TITLE>
<SERVER>

if (project.livepay == null)
{
project.livepay = registerNativeFunction("registerLivePayment",
                    "/usr/ns-home/bin/https/libccp.so",
                    "registerLivePayment", "true");
}

</SERVER>
</HTML>
```

In this example, “project.livepay” is a Boolean property of the **project** object, which indicates whether the function **registerLivePayment** has been successfully registered for the LivePayment objects. The **registerLivePayment** function registers the LivePayment objects, and “/usr/ns-home/bin/https/libccp.so” is the full path of the LivePayment shared library on the system. If you include “true” as the fourth parameter, you do not need to register the LivePayment functions on each page that calls them. If you do not include “true” or if you use “false” instead, you have to register the LivePayment functions on each page using **registerLivePayment**.

The following example demonstrates registering the LivePayment objects on the page that calls them:

```
<HTML>
<SERVER>

if (project.livepay == "true") //all properties of the project object
                               //are strings
{
result = registerLivePayment();
if (!result)
     write("Register LivePayment objects failed");
}
else
{
//
// You can start using LivePayment objects in this page now
//
}

</SERVER>
</HTML>
```

This example registers the LivePayment objects for this page and gives an error message if the registration is not successful.

Creating instances of LivePayment objects

The objects in LivePayment are predefined object types that have specific methods and properties. You create object instances of these object types using the **new** operator.

For example, to create an instance of a **Merchant** object, the syntax might be:

```
merchant = new Merchant ("00002650999", "Netscape")
```

This creates an instance of the **Merchant** object type called “merchant” and includes parameters for the merchant number and merchant name.

If you provide the correct parameters to create an object, you should not get any errors. However, you can check that the object was created successfully using the error status methods. For more information, see “Using error status methods” on page 150.

Some methods create and return objects. For example, the method **getCurrent-Batch** creates and returns a **Batch** object representing a current batch:

```
batch = processor.getCurrentBatch(terminal, merchant)
```

This method returns a **Batch** object for the current batch and the given processor, terminal, and merchant.

Using LivePayment properties

Each object type has predefined properties associated with it. They are either *optional* or *required*. You do not need to set an optional property. A required property must be set before you can invoke a particular method for the object, or before you can use the object as a parameter for a method. You can either set properties when you create the object, or set them by direct assignment. For example, the example that creates a Merchant object:

```
merchant = new Merchant ("00002650999", "Netscape")
```

sets the **merchantNumber** and **name** properties when the object is created. You could also set the property directly, for example:

```
merchant.merchantNumber = "00002650999"
```

In addition, you can use default property values for some properties. For more information, see “Using the default values from the LivePayment configuration” on page 149.

Property Types

Though properties of LivePayment objects can have types of number, Boolean, etc., properties of LiveWire’s predefined objects (for example, the **project** object) are *always* strings. You can assign numerical or Boolean values to these properties, but they are always converted and stored as strings; to retrieve such values, you must convert them back from strings. The **parseInt** and **parseFloat** functions are useful for converting to integer and floating point values. For more information, see the LiveWire documentation.

Numeric Strings

If FDC is your acquirer, when using LivePayment object properties that are the numeric strings, FDC expects to receive characters in the set "0" to "9" only. You cannot include decimal and sign (+, -) characters in a numeric string. For example, "-13.54" is not a numeric string that can be used as the value for the merchantReference property of the PayEvent object.

Using the default values from the LivePayment configuration

Some values for properties used by the objects are based upon the LivePayment configuration. Once set, your application automatically defaults to the values in the configuration for the following items:

- Merchant name
- Merchant number
- Terminal number
- Password file for slip encryption

For example, if you are creating a **Terminal** object, you usually create it using the **terminalNumber** property. However, if you set the default configuration for the terminal number, you can create the **Terminal** object without the **terminalNumber** property. It will pick up the default value for the terminal number from the configuration.

For example:

```
terminalObject= new Terminal("terminalNumber")
```

This object can instead use the default value for *terminalNumber*:

```
terminalObject= new Terminal()
```

For more information on configuring the defaults, see Chapter 2, “Setting up Netscape LivePayment”.

Using error status methods

Most methods return a value or an object. For methods that return a Boolean value or an object, the return value indicates the success or failure of the method invocation: true indicates success and false indicates failure. For methods that return an object, a non-null object indicates success, a null value indicates failure.

In addition, all objects have the following methods to indicate whether the object invocation has been successful so far:

Method	Return Value Type	Description
good	Boolean	Checks the status. Returns true if no error status has been set for the object. Otherwise, it returns false.
bad	Boolean	Checks the status. Returns true if at least one error status has been set for the objects. Otherwise, it returns false.
getStatusCode	String	Returns the status code, which is a string designating the latest error. If no status code has been set for this object, returns null.

Method	Return Value Type	Description
getStatusMessage	String	Returns a more detailed message or messages (multiple messages occur if there are multiple errors). It does not include any HTML tags. If no status message has been set for this object, returns null.
clearStatus	Void	Clears the status code and messages. Until you clear the status code and messages, more recent status codes and messages are appended to the existing status codes and messages (most recent first).

For example, the following code checks for errors using the above methods:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (processor.bad())
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

The example uses the **bad** method to test for an error. If **bad** is true, it writes "get current batch failed", gets the status code with **getStatusCode**, and gets the code's message with **getStatusMessage**. After writing the status code and status message, the application clears the status code and status message with **clearStatus**.

You can also check for errors by checking the return value. For example:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (batch == null)
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

A successful **getCurrentBatch** returns a **Batch** object. A null value indicates that **getCurrentBatch** was unsuccessful. The program tests for a null value, and, if it finds it, writes the message “get current batch failed.”

Getting the LivePayment version in LiveWire

If you need to know exactly which version of LivePayment you are using, you can find out using the **LivePaymentVersion** function. You use this function after you have registered the LivePayment objects. The function returns a string with the version in it.

For example:

```
<SERVER>

var livePaymentVersion = LivePaymentVersion();
write(livePaymentVersion);

</SERVER>
```

Using LivePayment objects to process payments

This section describes how to create the objects and use the methods for a LivePayment credit card processing application.

Creating Merchant and Terminal objects

The first thing you should do when starting to process credit card transactions is create the **Merchant** and **Terminal** objects. The merchant is the party doing business on the Internet, and the terminal is a device on which card processing takes place. Creating these objects requires merchant number and terminal number information from the bank card acquirer.

The **Merchant** object has the following properties:

Property	Type	Required/ Optional	Description
merchantNumber	String	Required	The merchant number, which is provided by the acquirer.
name	String	Required	The name of the merchant company.

The **Terminal** object has the following property:

Property	Type	Required/ Optional	Description
terminalNumber	String	Required	The terminal number, which is provided by the acquirer.

To create objects, you use the standard JavaScript operator **new**. To create a new **Merchant** object, you provide the values for the properties **merchantNumber** (which you get from your acquirer) and **name** (the name of your company).

To create a new **Terminal** object, you provide the **terminalNumber** property (which you get from your acquirer).

The following sample code creates a **Merchant** object "mer", and a **Terminal** object "term":

```
// create a Merchant and Terminal object
mer = new Merchant("00002650999", "Netscape");
term = new Terminal("00003277999");
```

If you have set up defaults in the LivePayment configuration for the merchant name, merchant number, and terminal number, you can also define the objects without specifying those values:

```
mer = new Merchant();
term = new Terminal();
```

Creating a Processor object

The **Processor** object contains information about the acquirer. **Processor** methods communicate with the acquirer through the gateway.

The **Processor** object has the following properties:

Property	Type	Required/ Optional	Description
encryptPasswordFile	String	Required	The name of the file that contains the password for the slip encryption.
name	String	Required	The name of the acquirer.

The **Processor** object has the following methods:

Method	Description
authorize	Authorizes a payment.
capture	Captures a payment.
credit	Credits an amount to a credit card.
getCurrentBatch	Gets the current batch number from the acquirer and creates a Batch object.
settleBatch	Settles a batch for the processor.

The following sample code creates a **Processor** object:

```
// create a Processor object
processor = new Processor("FDC", "encryptPassword.txt");
```

To create a **Processor** object, you need to know the name of your acquirer (in this case FDC) and the file that contains the password for the slip encryption key. You can also use the default in the LivePayment configuration for the password file and give only the acquirer name.

```
// create a Processor object
processor = new Processor("FDC");
```

Creating a Batch object

There are two ways to create an instance of a **Batch** object. The first is to use **getCurrentBatch**, which returns the current batch from the acquirer. The second is to use the **new** operator. You have to use **getCurrentBatch** initially, because that method returns the batch number. However, if you store the batch number, in subsequent cases when you need to create a **Batch** object you can use the stored batch number and create the object using the **new** operator.

The Batch object has the following properties:

Property	Type	Required/ Optional	Description
batchNumber	String	Required	The batch number returned by **getCurrent-Batch**. This number comes from the acquirer.
creditCount	Number	Required for **settleBatch**	The total number of credits in the batch.
currency	String	Required for **settleBatch**	The three-character ISO 4217 currency code. Some common codes: USD U.S. dollar CAD Canadian dollar FRF French franc For the complete list of currency codes, contact the International Standards Organization.
merchantReference	String	Required for **settleBatch**	Reference information provided by the merchant for tracking purposes. For FDC, the merchant reference cannot be more than ten alphanumeric characters.
salesCount	Number	Required for **settleBatch**	The total number of sales in the batch.

Property	Type	Required/ Optional	Description
totalSalesAmount	String	Required for **settleBatch**	The total sales amount of all credit card transactions in the batch.
totalCreditAmount	String	Required for **settleBatch**	The total refund/credit amount of all credit card transactions in the batch. Must be a positive number.

To create a **Batch** object with the **new** operator, the only property you need to set is the **batchNumber** property. The other properties are required for settling the batch.

To find out the current batch from your acquirer, you must first have created the **Terminal** and **Merchant** objects. Each batch is identified by a batch number, which your acquirer supplies in response to **getCurrentBatch**.

The following sample code creates a **Batch** object using **getCurrentBatch**, checks for errors, and stores the batch number:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check the errors
if (processor.bad())
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}

// remember the batchNumber or store in a database
batchNumber = batch.batchNumber;
```

This example shows getting the current batch number with the **Processor**, **Terminal** and **Merchant** objects created as “processor”, “term”, and “mer”. It checks for errors and sends messages if the batch was not opened successfully. If successful, **getCurrentBatch** returns a Batch object with the batchNumber property assigned by the acquirer.

The following example shows creating a Batch object using **new** operator. The **getCurrentBatch** method was run previously.

```
// Create the Batch object using the
// batch number of the batch which was defined earlier
batch2 = new Batch(batchNumber);
```

The **Batch** object created is called “batch2.”

Creating a Slip object

The **Slip** object contains the buyer’s credit card information, plus some optional information.

The **Slip** object has the following properties:

Property	Type	Required/ Optional	Description
amount	String	Required	The amount of money in the smallest unit. Only numeric characters are allowed (without a decimal point). The unit is based on the currency. For the U.S. dollar the unit is a cent.
billingStreet	String	Optional	The billing street address of the credit card holder. Must be less than 40 characters; additional characters are truncated.
billingZip	String	Optional	The billing zip code of the credit card holder.
cardExpiration	String	Required	The card expiration date (yyyymm).
cardNumber	String	Required	The credit card number.
cardType	String	Required	The credit card type. Valid types are Visa, MasterCard, AmericanExpress, Discover, JCB, DinersClub, and Carte-Blanche.

Property	Type	Required/ Optional	Description
currency	String	Required	The three-character ISO 4217 currency code. Some common codes: USD U.S. dollar CAD Canadian dollar FRF French franc For the complete list of currency codes, contact the International Standards Organization.
merchantReference	String	Optional	The merchant reference information. For example, an invoice number. Must be less than or equal to 16 alphanumeric characters.
purchaseRequestTime	string	Set automatically by Slip object	The time the slip was created. This field is set automatically by the **Slip** object and cannot be set manually. Read-only.

Once the slip has been encoded, its properties cannot be updated. Most properties cannot be accessed. A few are read-only:

- **merchantReference**
- **purchaseRequestTime**
- **cardType**

The **Slip** object has the following methods:

Method	Description
appendMerchantOrderDesc	Updates the merchant's description of the order. The merchant's order description must match the customer's order description. The description can be appended many times and does not need to be appended all at once. However, the sequence of the append must be the same as the append of the customer order description. You must finish appending to the order description before you authorize.
appendOrderDesc	Updates the customer's order description. The description can be appended many times and does not need to be appended all at once. You should not use **appendOrderDesc** after using **encode**.
encode	Encodes the data in the Slip object into an encrypted DER coded string.
getDER	Gets the printable ASCII encoded information from the slip DER.
initMerchantOrderDesc	Initializes the merchant's order description with the slip amount and currency. Use this method before using **appendMerchantOrderDesc**.

The following example shows how to create a **Slip** object at the merchant site. The example is in U.S. dollars:

```
// Create and generate a slip for max amount of 100.00
cardNumber = "5200000000000007";
cardExpiration = "199612";
currency = "USD";
maxAmount = "10000";
slip = new Slip(cardNumber, cardExpiration, maxAmount, currency);
```

The required information about the card and the order, including the card number, expiration date, amount, and currency, comes from the customer. This information is used to define the properties. Then you create a slip using that information. You use the **new** operator to create the **Slip** object.

Next you can set other fields in the slip. The following example shows setting fields for a slip called “slip”:

```
// set other fields before generating the encrypted slip
slip.cardType   = "MasterCard";
slip.merchantReference = "invoice 2789";
slip.billingStreet = "1234 Easy Street";
slip.billingZip = "94043";

slip.appendOrderDesc("Netscape Navigator Gold 2.0");
slip.appendOrderDesc("satisfaction guaranteed");
```

The order description and the merchant order description

A slip contains two order descriptions, the order description that a customer sees and the merchant order description. These must be the same, or the authorization fails. The purpose of the order descriptions is to assure the customer and the merchant agree upon what is ordered.

At the time the customer places orders, the application should display an exact description of the order including the amount and currency. Use **appendOrderDesc** to add the customer’s order description to the slip.

You can put multiple lines of information in the order description, and you can append to it a number of times.

Initialize the merchant’s order description with **initMerchantOrderDesc**. You can add additional information with **appendMerchantOrderDesc**.

Encoding and decoding a Slip object

After creating a slip, you encode it for security reasons. Once you encode the slip, you cannot set any more properties for it, so it’s important to define all the information you need before you encode the slip. After you encode the slip many properties are not accessible (card number, etc.) and the properties that are accessible (**merchantReference**, **purchaseRequestTime**, and **cardType**) are read-only.

The **encode** method puts the slip information in *DER (Distinguished Encoding Rules)* encoded string format. After encoding, you use the **getDER** method to get the encoded string. The example below encodes the slip and extracts it (still in DER format).

```
// encode the slip and
// extract the slip in DER format
slip.encode(processor);
asciiDER = slip.getDER();
```

You can then take the encoded string and recreate the slip from it. However, the properties will either be inaccessible or read-only. Inaccessible properties return null when you try to access them. The following example recreates the slip from the DER and calls it “slip2.” When the slip is recreated, you are able to access read-only properties.

```
// recreate the slip from the ascii DER
slip2 = new Slip(asciiDER);
```

You can access some slip information; however, some information is not accessible for security reasons. The following example demonstrates attempts to access inaccessible slip data:

```
// some data in the slip is not accessible
write(slip2.cardNumber == null);   // should print true
slip2.cardNumber = "23456";        // should get an error
slip2.getStatusCode();             //prints the error status code
slip2.getStatusMessage();          //and message
slip2.clearStatus()
```

You cannot read the card number, so the first statement returns null and prints true. The second line, which attempts to update the card number, returns an error.

Even though all other slip data cannot be updated, you can still enter a merchant order description. The merchant order description is stored separately from the slip. However, you must finish appending to the order description before you authorize a payment.

This example shows the order description being initialized with the amount and currency from the slip. You *must* use the slip's amount and currency. After initializing the description, this example appends some descriptive lines. The lines you add much match the order description entered when the slip was created.

```
// To provide the order description from the merchant
slip2.initMerchantOrderDesc("10000", "USD");
slip2.appendMerchantOrderDesc("Netscape Navigator Gold 2.0");
slip2.appendMerchantOrderDesc("satisfaction guaranteed");
```

Creating a PayEvent object

PayEvent objects contain the record of a complete financial transaction. For example, a **PayEvent** object can consist of a credit card authorization and capture, or a **PayEvent** object can be a credit.

The **PayEvent** object has the following properties:

Property	Type	Required/ Optional	Description
amount	String	Required for **authorize, capture,** and **credit**	The amount of money in the smallest unit. The amount must be greater than zero. Only numeric characters are allowed (without a decimal point). The unit is based on the currency. For the U.S. dollar the unit is a cent.
authCode	String	Required for **capture**	The authorization code, which is provided by the acquirer when the authorization is approved.

Property	Type	Required/ Optional	Description
avsResp	String	Required for **capture**	The address verification system response, which is provided by the acquirer when the authorization is approved. For example, if your acquirer is FDC, the system response is three-characters, where the first character represents the address match, the second character is the zip code match, and the third character is the authorizer verification result code. “Y” is a match, “N” is no match, and “X” is unavailable or incomplete service.
eventID	String	Required for **capture** and **credit**	The unique ID for a transaction within the batch. Used in capturing and crediting. The event ID is set by the merchant. The ID must be unique for the event; otherwise you will get an error. For FDC, the ID cannot be more than five alphanumeric characters.
eventTime	String	Set automatically after **authorize**, **capture**, and **credit**	The time the pay event occurred. Provided by the acquirer when the capture or credit is approved. Read-only.

Property	Type	Required/ Optional	Description
merchantReference	String	Required	Reference information the merchant assigns to the pay event. For example, the reference information might be the invoice number. For FDC, this reference cannot be more than ten numeric characters.
paySvcData	String	Required for **capture**	The payment service data or interchange compliant code, which is provided by the acquirer when the authorization is approved. Payment service data is only used for Visa and MasterCard.

For each **PayEvent** object, you need to define the merchant reference. The other **PayEvent** properties are set later when the object is passed to methods of the **Processor** object.

You create a **PayEvent** object using the **new** operator.

The following example creates a **PayEvent** object "pay":

```
// create a PayEvent object
MerchantReference = "0091";
pay = new PayEvent(MerchantReference);
```

Or you can assign the merchant reference value when you create the **PayEvent** object:

```
pay = new PayEvent("0091");
```

Authorizing a payment

If the pay event consists of an authorization and a capture, you first authorize the payment. To do that, you need to define the amount you want to authorize and the **eventID** property. In addition, you need the **Processor**, **Terminal**, **Merchant**, **PayEvent**, and **Slip** objects that you have previously created. This

example authorizes a purchase of $100.00 in U.S. dollars. It uses the previously created **Processor**, **Terminal**, **Merchant**, **PayEvent**, and **Slip** objects “processor”, “terminal”, “mer”, “pay”, and “slip”.

```
// To authorize for an amount of 100.00
pay.amount = "10000";
if (processor.authorize(terminal, mer, pay, slip))
{
     write(pay.authCode);
     write(pay.svcData);
     write(pay.avsResp);
     write(pay.eventTime);
}
else
{
     write("authorize failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

If the acquirer approves the authorization, the application returns the authorization code, the payment service data, the AVS response, and the time of the payment event. This example writes that information.

If the acquirer does not approve the authorization, or if some other error occurs, the example gives the user an error message, the status code, and the status message.

Capturing a payment

Once you have received verification of the authorization from your acquirer, you capture the amount of the sale. To capture a payment, use the **capture** method of the **Processor** object. You use the previously created **PayEvent**, **Batch**, **Terminal**, **Merchant**, and **Slip** objects. This example captures $98.50 in U.S. dollars. The **eventID** property is a unique number for the transaction in the batch.

```
// To capture an amount of 98.50
// authCode, svcData, and avsResp should be properly set in the PayEvent
// object (these values are returned by authorize)
pay.amount = "9850";
pay.eventID = "0001";
if (!processor.capture(terminal, mer, pay, slip, batch2))
{
     write("capture failed");
```

```
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
else
{
     write("capture succeeded");
     write(pay.eventTime);
}
```

If the capture succeeds, the application returns the time of the capture. If the capture fails, the application gives an error message, status code, and status message.

Crediting an account

To credit an account for a previously captured purchase, you need to know the payment data from the capture. You need to set an amount for the credit and a unique event ID. The following example sets these values and invokes the **credit** method:

```
// To credit an amount of 12.38
pay.amount = "1238";
pay.eventID = "0002";
processor.credit(terminal, mer, pay, slip, batch);
```

The event ID is a unique number for the transaction in the batch. In this example, “pay”, “processor”, “terminal”, “mer”, “pay” “slip” and “batch” are the previously created objects.

Settling payments

To settle payments, you settle the batch and check the total number of sales and credits and their total amounts against your records. The following example settles the batch and checks for errors.

```
// settle the batch
batch.merchantReference = "234234";
batch.totalSalesAmount = "340009";
batch.totalCreditAmount = "1238";
batch.salesCount = 37;
batch.creditCount = 1;
if (!processor.settleBatch(term, mer, batch))
```

```
{
     write("settle batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

If the batch cannot be settled successfully, the application gives an error message, the status code, and the status message.

Using the LPAuthOnly sample application

The LPAuthOnly sample application performs a simple function: it authorizes the purchase of a single item (a T-shirt) and writes the results to a logfile (/tmp/lpAuthOnly.log). There is no database involved, and it does not demonstrate the complete payment process. Instead, it provides a simple example of how to use Netscape LivePayment.

If you want to alter an existing application, we recommend that you begin with the Starter Application Set (LPStart and LPAdmin) that has database integration and the full range of LivePayment functions.

The LPAuthOnly files

The following files make up LPAuthOnly:

- auth.html—The file containing the JavaScript that authorizes a payment.
- build—The file that builds the .web file.
- error.html—An HTML page that displays error messages.
- home.html—The first page that the user sees to buy a T-shirt.
- purchase.html—The form for purchasing the T-shirt.
- start.html—The initial page where **registerLivePayment** is registered.

In addition, LPAuthOnly includes the following images:

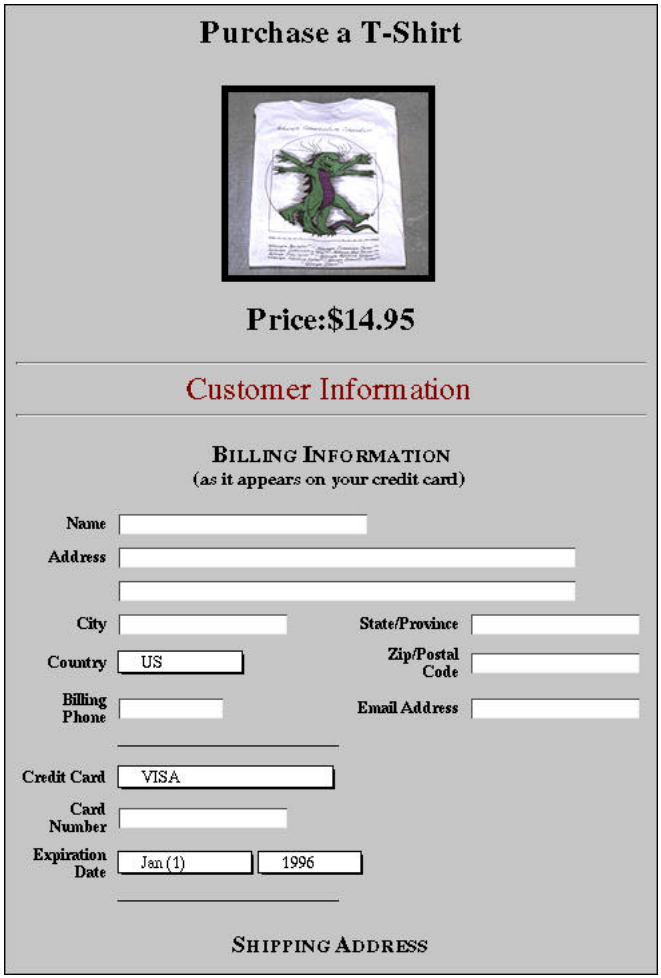
- tshirt.gif—The picture of the T-shirt.
- livepayment.gif—The picture of the LivePayment banner.

Running LPAuthOnly

The LPAuthOnly sample application is shipped with LivePayment and automatically installed when you install LivePayment. To use LPAuthOnly, from the Netscape LivePayment page, click the **LiveWire Application Manager** link. A list of applications appears. Click **LPAuthOnly** to display information about the application.

To run the application, with LPAuthOnly highlighted in the scrolling list, click **run** from the list of actions. A welcome page appears.

To purchase the T-shirt, click the **Purchase** link. The purchase.html page appears:



To try out the application, enter the information required and click **Place Order**. If successful, the application displays a receipt. If there is an error, the application displays the error on the error page (error.html).

As you can see, with LPAuthOnly you can only buy one item (the T-shirt) at one price ($14.95), and you can only buy one at a time. These assumptions are listed (among others) at the beginning of the auth.html file.

The sample code

The bulk of the sample code for credit card processing is in auth.html. The other files are either HTML pages for the user to see (home.html, error.html) or files that perform functions unrelated to card processing (build, start.html).

The file auth.html defines several functions and follows the basic LivePayment steps for authorization:

1. **Register the LivePayment objects for the page.**

   The application uses **registerLivePayment** to register the LivePayment objects for the page.

2. **Create a Merchant object.**

   The application creates a **Merchant** object using the default values supplied in the configuration. The configuration defaults are listed in the assumptions section of the application.

3. **Create a Terminal object.**

   The application creates a **Terminal** object using the default values supplied in the configuration. The configuration defaults are listed in the assumptions section of the application.

4. **Create a Processor object.**

   The application creates a **Processor** object using the default password file supplied in the configuration and "FDC" as the processor name. The configuration defaults are listed in the assumptions section of the application.

5. **Generate a Slip object.**

   The application uses the function **GetCurrentSlip** to generate the **Slip** object. This function creates and encodes the **Slip** object.

6. **Create a PayEvent object.**

   The application creates a **PayEvent** object first by calling a function **GetMerchantReference** and then by using the **new** operator to create the object.

7. **Authorize the transaction.**

   The application uses the **authorize** method of the **Processor** object to authorize the payment.

8. **Check the AVS.**

   The application uses the function **CheckAVS** to check the AVS result returned by the authorization. If both characters (address and zip) match, then the application accepts the authorization.

9. **Log transactions to a file.**

   To log the transactions to the lpAuthOnly.log file, the application uses the **Log** function. It creates a new file in a temporary directory and writes the transaction information to it. It checks whether the authorization succeeded or failed.

10. **Print the receipt or the error message.**

    If the authorization is successful, the application uses **PrintReceipt** to print the receipt. If the authorization fails, the application uses **PrintError** or **PrintFmtError**.

Chapter

9

LivePayment object reference

This chapter contains reference information for LivePayment's objects, properties, methods, and functions.

amount (PayEvent object)

Property. The amount of money in the smallest unit.

Syntax *payEventObjectName*.amount

Property of **PayEvent**

Description The amount must be greater than zero. Only numeric characters are allowed (without a decimal point). The unit is based on the currency. For the U.S. dollar the unit is a cent.

The type is string.

The property is required for **authorize**, **capture**, and **credit**.

Examples The following example shows the slip amount set to $14.95 in US dollars when the slip is generated. Later the same amount is used to set the amount property for the **PayEvent** object.

```
//
// Create a payevent object
//
```

```
var merchantReference = GetMerchantReference();
payevent              = new PayEvent(merchantReference);
payevent.amount       = "1495";
```

amount (Slip object)

Property. The amount of money in the smallest unit.

Syntax *slipObjectName*.amount

Property of **Slip**

Description Only numeric characters are allowed (without a decimal point). The unit is based on the currency. For the U.S. dollar the unit is a cent.

The type is string.

The property is required.

Examples See the example for **amount (PayEvent object)**.

appendMerchantOrderDesc

Method. Appends information to the merchant's description of the order.

Syntax *slipObjectName*.appendMerchantOrderDesc("*description_information*")

Parameters *slipObjectName* is the previously created slip.

description_information represents the order description information the merchant enters.

Method of **Slip**

Description Before using this method, you need to use **initMerchantOrderDesc**. The goal of **initMerchantOrderDesc** and **appendMerchantOrderDesc** is for merchants to provide their version of the order description. The important things to remember about **appendMerchantOrderDesc** are:

- The merchant order description must match the customer order description, which is entered with the **appendOrderDesc** method of the **Slip** object. The **authorize** method verifies that the two match.

- You can append to the merchant order description multiple times, as long as you append the same information as in the customer order description in the same sequence. For example, you might add the customer order description all at once, but for the merchant order description you might break the information up and append multiple times.
- You must finish appending to the order description before you authorize.
- If the description contains new line characters, those need to be included explicitly.

Returns true if successful, false if unsuccessful.

Examples The following example shows two lines updating the merchant order description:

```
slip.appendMerchantOrderDesc("Netscape Navigator Gold 2.0");
slip.appendMerchantOrderDesc("satisfaction guaranteed");
```

These two lines are added to the description for the slip "slip".

If you want to include a new line character, you must specify it as follows:

```
slip.appendMerchantOrderDesc("line\n");
```

See also **initMerchantOrderDesc**, **appendOrderDesc** methods

appendOrderDesc

Method. Appends lines to the customer's order description.

Syntax *slipObjectName*.appendOrderDesc("*description_information*");

Parameters *slipObjectName* is the previously created **Slip** object.

description_information represents the order information the customer enters.

Method of **Slip**

Description You can append to the description many times (before using **encode**), and you do not need to do all appends at the same time. If the description contains new line characters, those need to be included explicitly. You should not use **appendOrderDesc** after using **encode**.

Returns true if the append was successful, false if the append was unsuccessful.

Examples This example shows two lines of description added to the customer's order description.

```
slip.appendOrderDesc("Netscape Navigator Gold 2.0");
slip.appendOrderDesc("satisfaction guaranteed");
```

If you want to include a new line character, you must specify it as follows:

```
slip.appendOrderDesc("line\n");
```

See also **appendMerchantOrderDesc** method

authCode

Property. The authorization code, which is provided by the acquirer when the authorization is approved.

Syntax *payEventObjectName*.authCode

Property of **PayEvent**

Description The **authCode** type is string.

The property is required for **capture**.

Examples The following example shows the construction of a **PayEvent** object, which includes setting the **authCode**.

```
//
// construct PayEvent object
// note, purchaseCursor is a database cursor that has
// opened and located the to be authorized purchase record.
// terminal, merchant, payevent, slip, batch are objects
// which get created previously.
// The authCode along with other data from previous authorize()
// is picked up from the database and gets set in the PayEvent
// object before the capture.
//

merchantReference = slip.merchantReference;

payevent = new PayEvent(merchantReference);
payevent.amount = purchaseCursor.amount;
payevent.authCode = purchaseCursor.authCode;
payevent.paySvcData = purchaseCursor.paySvcData;
payevent.avsResp = purchaseCursor.avsResp;
eventID = GetNextEventID();
payevent.eventID = eventID;
```

```
//
// capture the payment
//
if (processor.capture(terminal, merchant, payevent, slip, batch))
     {
     purchaseCursor.eventID = eventID;
     purchaseCursor.status = "CAPTURED";
     if ((error = purchaseCursor.updateRow("crPurchase")))
     {
          purchaseCursor.close();
          database.rollbackTransaction();
          write("Failed to update crPurchase in database, error ",
                         error);
     }
}
```

authorize

Method. Authorizes a payment.

Syntax *processorObjectName*.authorize(*terminalObjectName, merchantObjectName, payEventObjectName, slipObjectName*)

Parameters *processorObjectName* is the previously created **Processor** object.

terminalObjectName, *merchantObjectName*, *payEventObjectName*, and *slipObjectName* are the previously created objects.

Method of **Processor**

Description Before authorizing a payment, you need to set the **amount** property of the **PayEvent** object. The authorization also checks the merchant's order description against the customer's order description to be sure they match.

If successful, returns true. The **authCode**, **paySvcData**, **avsResp** and **eventTime** in the **PayEvent** object are also set. If unsuccessful, returns false. If you are using loopback mode, the AVS that is returned is a random response so that if you check the AVS, you can test with a variety of responses.

Examples The following example shows the **authorize** method:

```
processor.authorize(terminal, mer, pay, slip);
```

Where "processor", "terminal", "mer", "pay", and "slip" are the previously created objects.

avsResp

Property. The address verification system result, which is provided by the acquirer when the authorization is approved.

Syntax *payEventObjectName*.avsResp

Property of **PayEvent**

Description If your acquirer is FDC, the system response is three characters, where the first character represents the address match, the second character is the zip code match, and the third character is the authorizer verification result code. “Y” is a match, “N” is no match, and “X” is unavailable or incomplete service.

Note that the acquirer still authorizes a payment even if the AVS does not match. It is up to the merchant to decide whether or not to continue with the transaction.

The type is string.

Required for capture.

Examples The following function checks the AVS response:

```
function CheckAVS(payevent)
{
     var avsfail = 0;

     avs = payevent.avsResp;
     if (avs.substring(0, 1) != "Y")
     {
           avsfail = 1;
     }

     if (avs.substring(1, 2) != "Y")
     {
          avsfail = 2;
     }
     return (avsfail);
}
```

bad

Method. Checks the error status to determine if object has been successfully created.

Syntax *ObjectName*.bad()

Parameters *ObjectName* is the object for which you are checking errors.

Method of All LivePayment objects.

Description Checks the status. Returns true if at least one error status has been set for the objects. Otherwise, it returns false.

Examples The following example checks to see if **getCurrentBatch** is successful:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (processor.bad())
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

See also **good**, **getStatusCode**, **getStatusMessage**, and **clearStatus** methods.

Batch

Object. Groups all credit card transactions that take place through a specific terminal until the batch is settled.

Syntax *batchObjectName* = new Batch("*batchNumber*")

Parameters *batchObjectName* is the Batch object you are creating.

batchNumber is the batch number returned by the previous **getCurrentBatch**. You can also create a **Batch** object by running **getCurrentBatch**. However, since **getCurrentBatch** always has to communicate across the Internet with the acquirer and return the number, it is more efficient to track the batch number and use it when you need to create a **Batch** object.

Description The **Batch** object groups all credit card transactions that take place through a specific terminal until the batch is settled. If your acquirer is FDC, there is always a current batch ready to receive your credit card transactions. When you settle one batch, FDC assigns a new current batch. You need to track the total sales amount and count and the total credit amount and count for the current batch. When you settle the batch, your totals for the credit card transactions are compared with the acquirer's. If they agree, the batch is settled.

Properties

- **batchNumber**
- **creditCount**
- **currency (Batch object)**
- **merchantReference (Batch object)**
- **salesCount**
- **totalSalesAmount**
- **totalCreditAmount**

Methods

- None

Examples The following example creates a **Batch** object called “batch2”:

```
batch2 = new Batch("00157");
```

See also **getCurrentBatch** method

batchNumber

Property. The batch number returned by **getCurrentBatch**.

Syntax *batchObjectName*.batchNumber

Property of **Batch**

Description This number comes from the acquirer when you run **getCurrentBatch**. If the current batch number is stored by the application, this property can also be assigned when a **Batch** object is created using the **new** operator.

The type is string.

The property is required.

Examples The following example creates a **Batch** object called “batch2” using a batch number “00157”:

```
batch2 = new Batch("00157");
```

The batch number in this example was previously returned by **getCurrent-Batch**.

See also **Batch** object, and **getCurrentBatch** method

billingStreet

Property. The billing street address of the credit card holder.

Syntax *slipObjectName*.billingStreet

Property of **Slip**

Description The billing street address must be less than 40 characters. Additional characters are truncated.

The type is string.

The property is optional.

Examples
```
slip.billingStreet = "1234 Easy Street";
```

See also **billingZip**

billingZip

Property. The billing zip code of the credit card holder.

Syntax *slipObjectName*.billingZip

Property of **Slip**

Description The **billingZip** property type is string.

The property is optional.

Examples
```
slip.billingZip = "94043";
```

See also **billingStreet**

capture

Method. Captures a payment.

Syntax
```
processorObjectName.capture(terminalObjectName, merchantObjectName,
     payEventObjectName, slipObjectName, batchObjectName)
```

Parameters *processorObjectName* is the previously created object.

terminalObjectName, *merchantObjectName*, *payEventObjectName*, *slipObjectName*, and *batchObjectName* are the previously created objects.

Method of **Processor**

Description Before capturing, you need to set the **amount** and **eventID** properties of the **PayEvent** object. The **eventID** must be unique within the batch. The **authCode**, **svcData**, and **avsResp** properties of the **PayEvent** object (which are returned from a successful **authorize**) should also be set.

If successful, returns true and the **eventTime** property in the **PayEvent** object is set. If unsuccessful, returns false.

Examples The following example shows the **capture** method:

```
processor.capture(terminal, mer, pay, slip, batch);
```

In this example "processor", "batch", "terminal", "mer", "pay", and "slip" are the previously created objects.

cardExpiration

Property. The credit card expiration date.

Syntax *slipObjectName*.cardExpiration

Property of **Slip**

Description The credit card expiration date is in the form yyyymm.

The type is string.

The property is required.

Examples The following example shows the creation of a new **Slip** object. The credit card expiration date is "199612".

```
slip = new Slip("5200000000000007", "199612", "10000", "USD");
```

cardNumber

Property. The credit card number.

Syntax *slipObjectName*.cardNumber

Property of **Slip**

Description The **cardNumber** type is string.

The property is required.

Examples The following example shows the creation of a new **Slip**. The credit card number is “5200000000000007”.

```
slip = new Slip("5200000000000007", "199612", "10000", "USD");
```

cardType

Property. The credit card type.

Syntax *slipObjectName*.cardType

Property of **Slip**

Description Valid types are Visa, MasterCard, AmericanExpress, Discover, JCB, DinersClub, and CarteBlanche.

The type is string.

The property is required.

Examples

```
slip.cardType = "MasterCard";
```

clearStatus

Method. Clears the status code and messages.

Syntax *ObjectName*.clearStatus()

Parameters *ObjectName* is the object for which you are checking errors.

Method of All LivePayment objects.

Description Until you clear the status code and messages, more recent status codes and messages are appended to the existing status codes and messages (most recent first). All status codes and messages are cleared when you use **clearStatus**.

Examples The following example checks to see if **getCurrentBatch** is successful:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (processor.bad())
```

```
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

See also **good**, **bad**, **getStatusCode**, and **getStatusMessage** methods.

credit

Method. Issues credit.

Syntax *processorObjectName*.credit(*terminalObjectName, merchantObjectName, payEventObjectName, slipObjectName, batchObjectName*)

Parameters *processorObjectName* is the previously created object.

terminalObjectName, *merchantObjectName*, *payEventObjectName*, *slipObjectName*, and *batchObjectName* are the previously created objects.

Method of **Processor**

Description Before using **credit**, you need to set the **amount** and **eventID** properties of the **PayEvent** object. The **eventID** must be unique within the batch.

If successful, returns true and the **eventTime** property in the **PayEvent** object is set. If unsuccessful, returns false.

Examples The following example shows the **credit** method:

```
processor.credit(terminal, mer, pay, slip, batch);
```

In this example “processor”, “terminal”, “mer”, “pay”, “slip”, and “batch” are the previously created objects.

creditCount

Property. The total number of credits in the batch.

Syntax *batchObjectName*.creditCount

Property of **Batch**

Description The **creditCount** property’s type is number.

The property is required for **settleBatch.**

Examples The following example sets the **creditCount** for a batch called “batch.”

```
batch.creditCount = 3;
```

See also **totalCreditAmount**

currency (Batch object)

Property. The three-character ISO 4217 currency code.

Syntax *batchObjectName*.currency

Property of **Batch**

Description Some common currency codes are:

USD	U.S. dollar
CAD	Canadian dollar
FRF	French franc

For the complete list of currency codes, contact the International Organization for Standardization.

The type is string.

The property is required for **settleBatch**.

Examples

```
batch.currency = "USD";
```

currency (Slip object)

Property. The three-character ISO 4217 currency code.

Syntax *slipObjectName*.currency

Property of **Slip**

Description Some common currency codes are:

USD	U.S. dollar
CAD	Canadian dollar
FRF	French franc

For the complete list of currency codes, contact the International Organization for Standardization.

The type is string.

The property is required.

Examples The following example sets the **currency** property of the **Slip** object by defining the currency and using it to create the **Slip** object.

```
// generate a slip
//
amount    = "1495";
currency  = "USD";
CCNumber  = "5200000000000007";
CCExpDate = "199612"

slip = new Slip(CCNumber, CCExpDate,amount,currency);

write(slip.currency);
```

encode

Method. Encodes the data in the **Slip** object into an encrypted, DER-encoded string.

Syntax *slipObjectName*.encode*(processorObjectName)*

Parameters *slipObjectName* is the previously created Slip object.

processorObjectName is the previously created Processor object.

Method of **Slip**

Description Once the slip is encoded, many of the properties are not accessible. Others are read-only. The method **getDER** is the only method you can use on the data stored in the encoded slip, though you can still append to the merchant order description (which is stored separately from the slip).

Returns true if successful, false if unsuccessful.

Examples The following example shows encoding a previously created slip:

```
slip.encode(processor);
```

The previously created **Slip** object is “slip” and the **Processor** object is “processor”.

See also **getDER** method

encryptPasswordFile

Property. The name of the file that contains the password for the slip encryption and decryption.

Syntax *processorObjectName*.encryptPasswordFile

Property of **Processor**

Description You can set the **encryptPasswordFile** property in you application, or use the default value you set for this property in the LivePayment parameter configuration. For more information on configuring the parameters, see “Configuring the LivePayment parameters” on page 32.

The type is string.

The property is required.

Examples The following example shows the creation of a new **Processor** object called “processor”. The processor name is “FDC”, and the encryptPasswordFile is “encryptPassword.txt”:

```
processor = new Processor("FDC", "encryptPassword.txt");
```

eventID

Property. The unique ID within the batch.

Syntax *payEventObjectName*.eventID

Property of **PayEvent**

Description The event ID is used in capturing and crediting. The ID must be unique for the event, otherwise you will get an error. For FDC, the ID cannot be more than five alphanumeric characters.

The type is string.

The property is required for **capture** and **credit**.

Examples The following function gets the next event ID by incrementing the current event ID by 1.

```
function GetNextEventID()
{
     project.lock();
     project.eventID = parseInt(project.eventID) + 1;
```

```
     var nextID = parseInt(project.eventID);
     project.unlock();
     return nextID;

}
payevent.eventID = GetNextEventID();
```

The resulting **eventID** is used by the **PayEvent** object.

eventTime

Property. The time the pay event occurred.

Syntax *payEventObjectName*.eventTime

Property of **PayEvent**

Description The time the pay event occurred. Provided by the acquirer when the authorize, capture, or credit is approved. Read-only.

The type is string.

The property is set automatically after **authorize**, **capture**, and **credit**

Examples The following line of code display the **eventTime** after a successful capture.

```
//
// capture a payment
//
if (processor.capture(terminal, merchant, payevent, slip, batch))
{
     write("A successful capture at ", payevent.eventTime);
}
```

getCurrentBatch

Method. Gets the current batch number from the acquirer and creates a **Batch** object.

Syntax
```
batchObjectName = processorObjectName.getCurrentBatch
     (terminalObjectName, merchantObjectName)
```

Parameters *batchObjectName* is the created Batch object.

processorObjectName, *terminalObjectName*, and *merchantObjectName* are the objects that have been created previously.

Method of **Processor**

Description You can use this method multiple times for the same current batch. It returns the batch object with the same batch number until the batch is settled, when it returns a current batch object. After using **getCurrentBatch** you can store the batch number and use the stored number to recreate the batch object when you need to.

If successful, this method returns a **Batch** object with the **batchNumber** property assigned by the processor. If unsuccessful, it returns null.

Examples The following example creates a batch object "batch" using a Terminal object "term" and a Merchant object "mer":

```
batch = processor.getCurrentBatch(term, mer);
```

See also **Batch** object, **batchNumber** property

getDER

Method. Gets the encoded information from the slip stored in DER (distinguished encoding rules) format.

Syntax *asciiDER* = *slipObjectName*.getDER()

Parameters *asciiDER* is the DER-encoded slip.

slipObjectName is the **Slip** object.

Method of **Slip**

Description This method gets the encoded information from the slip DER. This method only gets the information that is accessible after the slip has been encoded.

Returns the string of the slip DER if successful.

Examples The following example shows **getDER** used on an encoded slip.

```
asciiDER = slip.getDER()
```

In this example, "asciiDer" is the DER-encoded slip and "slip" is the **Slip** object.

See also **encode** method.

getStatusCode

Method. Gets the status code of the most recent error.

Syntax *ObjectName*.getStatusCode()

Parameters *ObjectName* is the object for which you are checking errors.

Method of All LivePayment objects.

Description Returns the status code, which is a string designating the latest error. If no status code has been set for this object, returns null.

Examples The following example checks to see if **getCurrentBatch** is successful:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (processor.bad())
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

See also **good**, **bad**, **getStatusMessage**, and **clearStatus** methods.

getStatusMessage

Method. Returns a detailed status message or messages.

Syntax *ObjectName*.getStatusMessage)

Parameters *ObjectName* is the object for which you are checking errors.

Method of All LivePayment objects.

Description Returns a detailed status message or messages (multiple messages occur if there are multiple errors). It does not include any HTML tags. If no status message has been set for this object, returns null.

Examples The following example checks to see if **getCurrentBatch** is successful:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);
```

```
// check for errors
if (processor.bad())
{
     write("get current batch failed");
     write("Processor status code = ", processor.getStatusCode());
     write("Processor status message = ",processor.getStatusMessage());
     processor.clearStatus();
}
```

See also **good**, **bad**, **getStatusCode**, and **clearStatus** methods.

good

Method. Checks the error status to determine if object has been successfully created.

Syntax *ObjectName*.good()

Parameters *ObjectName* is the object for which you are checking errors.

Method of All LivePayment objects.

Description Checks the status. Returns true if no error status has been set for the object. Otherwise, it returns false.

Examples The following example checks to see if **getCurrentBatch** is successful:

```
// get the latest batch
batch = processor.getCurrentBatch(term, mer);

// check for errors
if (processor.good())
{
     write("current batch number is", batch.batchNumber);
}
```

See also **bad**, **getStatusCode**, **getStatusMessage**, and **clearStatus** methods.

initMerchantOrderDesc

Method. Initializes the merchant's order description.

Syntax *slipObjectName*.initMerchantOrderDesc("*amount*", "*currency*");

Parameters *slipObjectName* is the **Slip** object.

amount and *currency* are the values on the slip.

Method of **Slip**

Description Initializes the merchant's order description. The amount and the currency in the description should be identical to the ones on the slip. Also, the merchant order description must match the customer order description.

Returns true if successful, false if unsuccessful.

Examples In the following example, "slip2" is the **Slip** object:

```
slip2.initMerchantOrderDesc("10000", "USD");
```

See also **appendMerchantOrderDesc** and **appendOrderDesc** methods

livePaymentVersion

Function. Gets the current version of LivePayment.

Syntax

```
livePaymentVersion()
```

Parameters None.

Description Use this function to find out the version of LivePayment you are using.

Examples The following example shows finding the version and writing it:

```
var livePaymentVersion = LivePaymentVersion();
write(livePaymentVersion);
```

Merchant

Object. Represents a merchant doing commerce on the Internet.

Syntax You can create a **Merchant** object using two different syntax options. The first requires strings for the **merchantNumber** and **name** properties. The second requires no parameters, because it picks up the default merchant number and name from the configuration. This example shows the first syntax, with strings for the configuration information:

```
merchantObjectName = new Merchant("merchantNumber", "name")
```

This example shows the second syntax, which uses the default configuration information:

```
merchantObjectName = new Merchant()
```

Parameters *merchantObjectName* is the new Merchant object.

merchantNumber is the merchant number, which is provided by the acquirer. *name* is the name of the merchant company.

Description The **Merchant** object represents a merchant doing commerce on the Internet. The merchant establishes a relationship with an acquirer to perform credit card transactions through the acquirer. The merchant is identified by a merchant number, which comes from the acquirer.

Properties

- **merchantNumber**
- **name (Merchant object)**

Methods

- None

Examples The following examples show the two syntax options, depending upon whether you use the default configurations or not.

This example shows the creation of a new merchant "mer" with a merchant number of "00002650999" and a name of "Netscape":

```
mer = new Merchant("00002650999", "Netscape");
```

This example picks up the default configuration information:

```
mer = new Merchant();
```

merchantNumber

Property. The merchant number provided by the acquirer.

Syntax *merchantObjectName*.merchantNumber

Property of **Merchant**

Description You can set this property in you application, or use the default value you set for this property in the LivePayment parameter configuration. For more information on configuring the parameters, see "Configuring the LivePayment parameters" on page 32.

The type is string.

The property is required.

Examples The following examples show setting the **merchantNumber** property in the process of creating a new **Merchant** object.

The first example shows the creation of a new merchant “mer” with a merchant number of “00002650999” and a name of “Netscape”.

```
mer = new Merchant("00002650999", "Netscape");
```

The next example picks up the default configuration information.

```
mer = new Merchant();
```

merchantReference (Batch object)

Property. Batch reference information provided by the merchant

Syntax *batchObjectName*.merchantReference

Property of **Batch**

Description If your acquirer is FDC, the merchant reference cannot be more than ten alpha-numeric characters.

The type is string.

The property is required for **settleBatch**.

Examples The following example shows the merchant reference “1234” defined for a **Batch** object “batch.”

```
batch.merchantReference = 1234;
```

merchantReference (PayEvent object)

Property. PayEvent reference information provided by the merchant.

Syntax *payEventObjectName*.merchantReference

Property of **PayEvent**

Description Reference information the merchant assigns to the pay event. For example, the reference information might be the invoice number. For FDC, this reference cannot be more than ten numeric characters.

The type is string.

The property is required.

Examples The following example creates a **PayEvent** object called “pay” using a merchantReference value of “2001”:

```
pay = new PayEvent("2001");
```

merchantReference (Slip object)

Property. Slip reference information provided by the merchant.

Syntax *slipObjectName*.merchantReference

Property of **Slip**

Description The **merchantReference** property must be less than 16 alphanumeric characters.

The type is string.

The property is optional.

Examples

```
slip.merchantReference = "invoice 2789";
```

name (Merchant object)

Property. The name of the merchant’s company.

Syntax *merchantObjectName*.name

Property of **Merchant**

Description You can set the name property in your application or use the default value you set for this property in the LivePayment parameter configuration. For more information on configuring the parameters, see “Configuring the LivePayment parameters” on page 32.

The type is string.

The property is required.

Examples This example shows the creation of a new merchant “mer” with a merchant number “00002650999” and name “Netscape”:

```
mer = new Merchant("00002650999", "Netscape");
```

name (Processor object)

Property. The name of the acquirer.

Syntax *processorObjectName*.name

Property of **Processor**

Description The name of the acquirer.

The type is string.

The property is required.

Examples The following example creates a **Processor** object called “processor” and sets the name property to “FDC”. It uses the default password file:

```
processor = new Processor("FDC");
```

PayEvent

Object. A particular event in a financial transaction.

Syntax *payEventObjectName* = new PayEvent("*merchantReference*")

Parameters *payEventObjectName* is the PayEvent object you are creating.

merchantReference is the merchant reference property.

Description For example, authorization and capture are two pay events that together make up a complete customer payment. Credit is another type of pay event.

Properties

- **amount (PayEvent object)**
- **authCode**
- **avsResp**
- **eventID**
- **eventTime**
- **merchantReference (PayEvent object)**
- **paySvcData**

Methods None.

Examples The following example creates a **PayEvent** object called “pay”:

```
pay = new PayEvent("2001");
```

“2001” is the merchantReference property.

paySvcData

Property. The payment service data or interchange compliant code, which is provided by the acquirer when the authorization is approved.

Syntax *payEventObjectName*.paySvcData

Property of **PayEvent**

Description The payment service data is only provided for MasterCard and Visa credit cards.

The type is string.

The property is required for **capture**.

Examples See the example for the **authCode** property.

Processor

Object. Represents the bank card acquirer that handles payment transactions.

Syntax There are two different syntax options for creating the new object. The first requires strings for the **name** and **encryptPasswordFile** properties. The second requires only the string for the **name** property. The default password file is picked up from the default configuration. This example shows the first syntax:

```
processorObjectName = new Processor("name","encryptPasswordFile");
```

This example shows the second syntax, which uses the default password file:

```
processorObjectName = new Processor("name");
```

Parameters *processorObjectName* is the new Processor object.

name and *encryptPasswordFile* are the processor name and the name of the file that contains the password for encryption.

Description The **Processor** object represents the bank card acquirer that handles payment transactions.

Properties

- **encryptPasswordFile**
- **name (Processor object)**

Methods

- **authorize**
- **capture**
- **credit**
- **getCurrentBatch**
- **settleBatch**

Examples The following examples show the two syntax options, depending upon whether you use the default values from the configuration or not. The first example does not use the default values:

```
processor = new Processor("FDC", "encryptPassword.txt");
```

In this example, the processor name is “FDC”, and the name of the file that has the encryption password is “encryptPassword.txt”.

The second example users the default password file:

```
processor = new Processor("FDC");
```

purchaseRequestTime

Property. The time the slip was created.

Syntax *slipObjectName*.purchaseRequestTime

Property of **Slip**

Description This property is set automatically by the Slip object and cannot be set manually. It is read-only.

The type is string.

Examples The following example creates a slip and writes the purchase request time.

```
//
// generate a Slip
//
if slip = new Slip("5200000000000007", "199612", "10000", "USD"));
```

```
{
     write("Slip created at ", slip.purchaseRequestTime);
}
```

registerLivePayment

Function. Registers LivePayment's objects with LiveWire.

Syntax `registerLivePayment()`

Parameters None.

Description If you do not use the true parameter on **registerNativeFunction** you must use **registerLivePayment** on every page that uses LivePayment's objects

Returns true if successful, false if unsuccessful.

Examples The following example registers the LivePayment objects in a page:

```
if (project.livepay == null)
{
result = registerLivePayment();
if (!result)
     write("Register LivePayment objects failed");
}
//
// You can start using LivePayment objects in this page now.
//
```

See also **registerNativeFunction** function

registerNativeFunction

Function. Registers LivePayment's **registerLivePayment** function and the shared library with LiveWire. The function should be invoked in the initial page of the LivePayment application

Syntax
```
registerNativeFunction("registerLivePayment", "library_path",
     "registerLivePayment", ["true"])
```

Parameters *library_path* is the pathname to the LivePayment shared library on your system.

Description You must register the **registerLivePayment** function and the shared library before you can use LivePayment's objects in your application.

If you set the fourth parameter to true, you do not need to invoke **registerLivePayment** on each page that calls the LivePayment objects. LiveWire automatically invokes **registerLivePayment** for every page accessed in the application. If you do not use set the fourth parameter, or if you set it to false, you will have to register the objects on each page that uses the LivePayment objects.

Returns true if successful, false if unsuccessful.

Examples The following example registers the **registerLivePayment** function in the initial page of your application.

```
if (project.livepay == null)
{
project.livepay = registerNativeFunction("registerLivePayment",
                    "server_root/bin/https/libccp.so",
                    "registerLivePayment", "true");
}
```

server_root is the directory where the server is installed.

See also **registerLivePayment** function

salesCount

Property. The total number of sales in the batch.

Syntax *batchObjectName*.salesCount

Property of **Batch**

Description The total number of sales in the batch.

The type is number.

The property is required for **settleBatch**.

Examples The following example totals the sales count and sales amount.

```
//
// Total up amount and number of transactions to settle
//
var totalSalesAmt = 0;
var salesCount = 0;

database.beginTransaction();
```

```
purchaseCursor = database.cursor("select sum(amount), count(distinct
ID) from crPurchase where batchID = " + batch.ID + " AND status =
\"CAPTURED\"");

if (purchaseCursor.next())
{
     totalSalesAmt = purchaseCursor[0];
     salesCount = purchaseCursor[1];
}
else
{
     purchaseCursor.close();
     database.rollbackTransaction();
     PrintError("Failed to get totalSalesAmt and salesCount for batch ",
          batch.ID);
}

purchaseCursor.close();

//
// Set up batch object
//
batch.merchantReference = batch.ID;
batch.totalSalesAmount = totalSalesAmt;
batch.totalCreditAmount = 0;
batch.salesCount = salesCount;
batch.creditCount = 0;
```

See also **totalSalesAmount** property

settleBatch

Method. Settles a batch for the processor.

Syntax *processorObjectName*.settleBatch(*terminalObjectName*, *merchantObjectName*, *batchObjectName*)

Parameters *processorObjectName* , *terminalObjectName*, *merchantObjectName*, and *batchObjectName* are the objects that have been created previously.

Method of **Processor**

Description Before settling a batch, you need to set the batch properties **currency**, **merchantReference**, **totalSalesAmount**, **totalCreditAmount**, **salesCount**, and **creditCount**.

Returns true if successful, false if unsuccessful.

Examples This example settles the batch and also gives you messages if **settleBatch** was unsuccessful. The objects in this example are “processor”, “term”, “mer”, and “batch”.

```
if (!processor.settleBatch(term, mer, batch))
{
    write("settle batch failed");
    write("Processor status code = ", processor.getStatusCode());
    write("Processor status message = ", processor.getStatusMessage());
    processor.clearStatus();
}
```

Slip

Object. Contains credit card information and order information.

Syntax There are two possible syntaxes for creating a slip. The first creates a **Slip** object:

```
slipObjectName = new Slip("cardNumber", "cardExpiration", "amount",
     "currency")
```

The second syntax recreates a slip from the encoded DER:

```
slipObjectName = new Slip("asciiDER")
```

Parameters For creating a new **Slip** object:

slipObjectName is the name of the slip.

cardNumber, *cardExpiration*, *amount*, and *currency* are the required properties.

For reconstructing the slip from the encoded DER:

slipObjectName is the name of the recreated slip.

asciiDER is the encoded DER in ASCII string, which is an opaque representation of the slip after the slip is encoded.

Description The slip contains credit card information and order information. The credit card data, such as the credit card number and expiration date, is encoded for security reasons. Some of the data cannot be accessed once encrypted. This encrypted information is used by the acquirer to approve and collect the payment for the merchant.

The slip includes two types of reference information: the customer order description and the merchant order description. The customer order description is supplied when the customer first enters the slip information. The merchant order description must match the customer description, but the merchant enters it later.

The merchant order description is stored separately from the **Slip** object, so it can be updated even after the **Slip** object has been encoded. To enter merchant order description information, the merchant uses **initMerchantOrderDesc** and **updateMerchantOrderDesc**. The merchant order description must be completely updated before **authorize**, when it is verified against the customer order description.

Properties

- **amount (Slip object)**
- **billingStreet**
- **billingZip**
- **cardExpiration**
- **cardNumber**
- **cardType**
- **currency (Slip object)**
- **merchantReference (Slip object)**
- **purchaseRequestTime**

Methods

- **appendMerchantOrderDesc**
- **appendOrderDesc**
- **encode**
- **getDER**
- **initMerchantOrderDesc**

Examples

The following example shows the creation of a new **Slip**:

```
slip = new Slip("5200000000000007", "199612", "10000", "USD");
```

The following example shows the creation of a slip called “slip2” from the encoded file “asciiDER”.

```
slip2 = new Slip("asciiDER");
```

Terminal

Object. A device on which card processing takes place.

Syntax There are two different syntax options. The first requires a string for the **terminalNumber** property. The second requires no parameters, because it picks up the default **terminalNumber** from the configuration. This example shows the first syntax, with a string for the terminal number:

```
terminalObjectName = new Terminal("terminalNumber");
```

This example shows the second syntax, which uses the default configuration information:

```
terminalObjectName = new Terminal()
```

Parameters *terminalObjectName* represents the terminal object.

terminalNumber represents the terminal number received from the acquirer.

Description You can have more than one terminal. It is the equivalent of having a checkout counter at a supermarket; if you have multiple checkers, you can have multiple counters. In the same way, you can have multiple terminals.

Properties

- **terminalNumber**

Methods None.

Examples The following examples show the two syntax options, depending upon whether you use the default configurations or not.

This example shows the creation of a new **Terminal** object with a terminal number of “00003277999”.

```
term = new Terminal("00003277999");
```

This example picks up the default configuration information.

```
term = new Terminal();
```

terminalNumber

Property. The terminal number provided by the acquirer.

Syntax *terminalObjectName*.terminalNumber

Property of **Terminal**

Description The terminal number provided by the acquirer. You can set this property in your application or use the default value you set for this property in the LivePayment parameter configuration. For more information on configuring the parameters, see “Configuring the LivePayment parameters” on page 32.

The type is string.

The property is required.

Examples The first example sets the **terminalNumber** property while creating the Terminal object “term”.

```
term = new Terminal("00003277999");
```

The following example creates the **Terminal** object “term” by picking up the default value from the configuration.

```
term = new Terminal();
```

totalCreditAmount

Property. The total amount of money of the credit transactions in the batch.

Syntax *batchObjectName*.totalCreditAmount

Property of **Batch**

Description The total credit amount (in the smallest unit) of all credit card transactions in the batch. Must be a positive number.

The type is string.

The property is required for **settleBatch**.

Examples

```
batch.totalCreditAmount = 4485
```

See also **creditCount** property

totalSalesAmount

Property. The total amount of all the sales in the batch.

Syntax *batchObjectName*.totalSalesAmount

Property of **Batch**

Description The total sales amount (in the smallest unit) of all credit card transactions in the batch.

The type is string.

The property is required for the **settleBatch** method.

Examples See the example for the **salesCount** property.

See also **salesCount** property

4

Creating an application using the cpcmd utility

- **Using the cpcmd utility**

Chapter

10

Using the cpcmd utility

This chapter provides an overview and description of the LivePayment **cpcmd** (card processor command) utility.

This chapter contains the following sections:

- Overview of cpcmd
- Using TraceFile and TraceLevel
- Command reference

Overview of cpcmd

The LivePayment **cpcmd** utility authorizes, captures, credits, and creates a credit card slip, as well as settles a batch containing credit card transaction information. You can use the command in a script, for example a shell script, to issue credit card transaction utility commands.

Chapter 3, “Payment application concepts” contains a detailed description of the transaction flow for credit card processing. You should read this chapter before creating your payment application. It also describes strategies for creating a successful credit card processing application. For example, it includes information on maintaining a state for the payment and the batch so that you can recover from system down time.

The **cpcmd** utility has several commands which process credit card transactions. The commands are:

Command	Description
Authorize	Authorizes a purchase. No money is charged to the credit card, though an amount is reserved pending a capture.
Capture	Captures a previously authorized purchase. This command charges the money to a customer's credit card.
CreateSlip	Creates a slip file of credit card data to be used by other commands. The information in this file is encrypted for security reasons.
Credit	Credits a purchase.
GetCurrentBatch	Gives you the number of the current batch for credit card transaction information.
SettleBatch	Settles a batch of credit card transaction information.

Transaction flow

The order in which you use these commands and which ones are required vary from situation to situation. The following description of the transaction flow may differ from your implementation. For a more detailed description of the credit card transactions, see “Credit card transactions” on page 52.

After you ask the customer for the credit card data, you use **CreateSlip** to create a slip. The data in the slip file is used by other credit card transactions.

Depending upon the requirements of your acquirer and your preference for organizing your data, you may run **GetCurrentBatch** next to start a batch.

When a customer indicates that he or she wants to make a purchase, run the **Authorize** command to check that the customer's account has the required amount of money available. This command also puts a hold on the amount of the purchase.

Once the purchase has been authorized, run the **Capture** command to inform the acquirer of the actual amount of money to be charged.

You can credit accounts using the **Credit** command.

If you are using batches, when you are ready to settle the batch use the **SettleBatch** command. The **SettleBatch** command compares totals you enter with totals contained in the batch and begins the transfer of funds between the merchant account and the card issuing banks.

Default configuration

Some values for arguments used by the utility commands are set during configuration. Once set, you can omit the arguments and the default values are used as your application runs.

You can set the defaults for the following items:

- Merchant number
- Terminal number
- Password file

For example, for a command that requires the **TermNum** and **MerNum** arguments you can either use the arguments and their values, or you can leave the arguments out and the utility will use the defaults in the configuration file. For example:

```
cpcmd -command getcurrentbatch -termnum 00003277999
-mernum 00002650999
```

can instead be sent with the default value for the terminal and merchant numbers:

```
cpcmd -command getcurrentbatch
```

For more information on configuring the defaults, see Chapter 2, “Setting up Netscape LivePayment”.

Data storage

You will probably want to use a database to store the transaction state and payment activity information of your application. If you are using a scripting language such as Perl, refer to the associated documentation for information about integrating database access.

Return values, output, and errors

The cpcmd utility returns 0 if it executes successfully, -1 if it does not. The commands send output (the batch number, authorization code, etc.) to stdout. Error messages go to stderr, unless you use the **tracefile** argument.

Using TraceFile and TraceLevel

For each of the **cpcmd** commands, you can also use the diagnostic arguments **TraceFile** and **TraceLevel**. These arguments are always optional.

Argument	Description
-TraceFile *trace_file*	The name of the file to which tracing information is sent.
-TraceLevel *trace_level*	The level of tracing that is sent to the trace file. This argument is used only if the **TraceFile** argument is used. The default value is "LevelEvent|ProcessID". Values include: Time — Timestamp each line of output. ProcessID — Output process ID to each line of output. DatedFile — The trace filename will be dated, and a new file created each day. LevelDebug — Output debugging level information. LevelEvent — Output the event. LevelDetail — Output detail information.

Output The output is the output of the command with which you are using **TraceFile**.

Example The following example shows an example of how to use **TraceFile** with the **GetCurrentBatch** command.

```
cpcmd -command getcurrentbatch -mernum 00002650999 -termnum 00003277999
-tracefile /tmp/cpcmd.trc

batch number 00023
```

This example shows a query of the open batch for a merchant number of 00002650999 and a terminal number of 00003277999. It outputs the trace information to the file /tmp/cpcmd.trc. Since no trace level was specified, the trace information defaults to level event and process ID.

The following example shows the /tmp/cpcmd.trc file:

```
11:21:20_23010: Argument Lists:
11:21:20_23010: ---------------
11:21:20_23010: Command = getcurrentbatch
11:21:20_23010: Processor = DefaultCardProcessor
11:21:20_23010: ConfigFile = system.ini
11:21:20_23010: ConfigDir = ../config
11:21:20_23010: TraceFile = /tmp/cpcmd.trc
11:21:20_23010: TraceLevel = LevelEvent|ProcessID
11:21:20_23010: Termnum = 00003277999
11:21:20_23010: Mernum = 00002650999
11:21:20_23010:
11:21:20_23010: Execute command GetCurrentBatch:
11:21:20_23010: created txsccp_c(00002650999, 00003277999) = 0xe8880
```

The numbers on the left margin indicate the time and the process ID.

Command reference

The following section describes the **cpcmd** commands in detail. They are listed in alphabetical order. For each command, the arguments are listed in alphabetical order.

A few guidelines for using the utility and this documentation:

- There must always be a space between the argument and its value.
- Arguments are not case sensitive. For example, LivePayment accepts either **TCreditAmt** or **tcreditamt**. However, the values of arguments are case sensitive. For example, an authorization code of A456789 is not the same as an authorization code of a456789. Names of commands are not case-sensitive (for example, **authorize** and **Authorize** both work).
- In the examples, the arguments are in mixed upper/lower case so that you can read them easily.

Authorize

The **Authorize** command authorizes a credit card for an amount of money. It reserves the amount authorized against the card's credit limit, but does not make a charge against the credit card.

Syntax **cpcmd -Command Authorize -MerNum** *merchant_number* **-TermNum** *terminal_number* **-Currency** *currency* **-SlipAmount** *slip_ amount* **-Amount** *amount* [**-SlipFile** *slip_file*] [**-MerchantRef** *merchant_reference*] **-OrdDescFile** *order_description_file* [**-PswdFile** *password_file*]

Arguments The following arguments are available for the **Authorize** command:

Argument	Type	Required/ Optional	Description
-Amount	Integer	Required	The amount of money. The unit is based on the currency code. For the US dollar the unit is a cent.
-Currency	Alpha	Required	The three-character ISO 4217 currency code. Some common codes: USD US dollar CAD Canadian dollar FRF French Franc For the complete list of currency codes, contact the International Standards Organization.
-MerchantRef	Alphanumeric	Optional	Reference information provided by the merchant for tracking purposes. If not specified, the default is the **MerchantRef** from the slip. Analogous to the **merchantReference** of the **PayEvent** object (if using LivePayment objects).
-MerNum	Alphanumeric	Required	The merchant number, which is provided by the acquirer.
-OrdDescFile	String	Required	The name of the file that contains the order description including type of goods ordered, price, delivery information etc.

Argument	Type	Required/ Optional	Description
-PswdFile	String	Optional	The filename of the password file containing the password used to encode/decode the slip. If not specified, the default is ../config/lp-default-pswdfile.
-SlipAmount	Integer	Required	The amount from the slip.
-SlipFile	String	Optional	The filename of the slip. If not specified, the default is cpslip.slp.
-TermNum	Alphanumeric	Required	The terminal number, which is provided by the acquirer.

Output If the authorization is successful, this command outputs a message saying the payment was authorized. In addition, it outputs the Authorization Code, the Payment Service Data, and the Address Verification Code. These values are required by the **Capture** command. and must be stored for use during capture.

Example The following example shows and example of the command followed by an example of the output:

```
cpcmd -command authorize -termnum 00003277999 -mernum 00002650999
-slipamount 1295 -amount 1295 -orddescfile ord.dsc -currency USD
```

```
Payment Authorized for USD1295
Authz code: 50TEST
Payment Svc data:MCC56789012345
AVS result: YYX
```

This example authorizes the amount of $12.95 (in US dollars). It uses the credit card information encoded in the file cpslip.slp. It references the order description file ord.dsc. It defaults to the slip's merchant reference.

It outputs the amount that was authorized, the Authorization Code, the Payment Service Data, and the Address Verification result. You may get a result of all blanks for the Payment Service Data depending upon the card type. Payment Service Data is only used for Visa and Mastercard. For more information on interpreting the AVS result, see “Results of authorize” on page 64.

Capture

The **Capture** command charges a customer for the amount authorized.

Once the transaction between the merchant and the customer is completed, the **Capture** command sends information back to the acquirer that authorized the transaction.

Syntax

cpcmd -Command Capture -MerNum *merchant_number*
-TermNum *terminal_number* **-Amount** *amount*
-AuthzCode *authorization_code* **-PaySvcData** "*payment_service_data* "
-AVS *address_verification_code* [**-SlipFile** *slip_file*]
[**-MerchantRef** *merchant_reference*] **-BatchNumber** *batch_number*
[**-PswdFile** *password_file*] **-TranxId** *transaction_ID*

The payment service data argument value is in quotes because the value of payment service data has a space in it.

Arguments

The following arguments are available for the **Capture** command:

Argument	Type	Required/ Optional	Description
-Amount	Integer	Required	The amount of money. The unit is based on the currency code. For the US dollar the unit is a cent.
-AuthzCode	Alphanumeric	Required	The authorization code, which is returned from the **Authorize** command.
-AVS	Alpha	Required	The address verification service result, which is returned from the **Authorize** command.
-BatchNumber	Integer	Required	The batch number generated by the **GetCurrentBatch** command.

Argument	Type	Required/ Optional	Description
-MerchantRef	Alphanumeric	Optional	Reference information provided by the merchant for tracking purposes. If not specified, the default is the **MerchantRef** from the slip. Analogous to the **merchantReference** of the **PayEvent** object (if using LivePayment objects).
-MerNum	Alphanumeric	Required	The merchant number, which is provided by the acquirer.
-PaySvcData	Alphanumeric	Required	The payment service data or interchange compliant code, which is returned from the **Authorize** command. Payment service data is only used for Visa and Mastercard.
-PswdFile	String	Optional	The filename of the password file containing the password used to encode/decode the slip. If not specified, the default is ../config/lp-default-pswdfile.
-SlipFile	String	Optional	The filename of the slip. If not specified, the default is cpslip.slp.
-TermNum	Alphanumeric	Required	The terminal number, which is provided by the acquirer.
-TranxId	Integer	Required	The transaction ID. It must be unique within the batch.

Output If successful, outputs the currency code and the amount captured.

Example

```
cpcmd -command capture -termnum 0003277999 -mernum 0002650999 -amount
1295 -authzcode 50TEST -paysvcdata “MCC56789012345 TEST0001” -AVS YYX
-batchnumber 00010 -tranxid 99912
```

```
captured USD1295
```

This example captures 12.95 in US dollars. It uses the authorization code, payment service data, AVS response returned by the authorization. It uses the default values for the password file and the slip file. It uses the merchant reference from the slip.

CreateSlip

The **CreateSlip** command creates the electronic equivalent of a paper credit card slip.

Syntax

cpcmd -Command CreateSlip -Currency *currency*
-SlipAmount *slip_amount* [**-ClientRef** *client_reference*]
[**-MerchantRef** *merchant_reference*] [**-SlipFile** *slip_file*]
[**-SlipExpDate** *slip_expiration_date*] **-CardType** *card_type*
-PAN *personal_account_number* **-PANExpDate** *PAN_expiration_date*
[**-OrdDescFile** *order_description_file*] [**-BillStreet** *billing_street*]
[**-BillZip** *billing_zip_code*] [**-PswdFile** *password_file*]

Arguments

The following arguments are available for the **CreateSlip** command:

Argument	Type	Required/ Optional	Description
-BillStreet	String	Optional	The billing street address of the credit card holder. You must enclose the value for this argument in quotes.
-BillZip	String	Optional	The billing zip code of the credit card holder. You must enclose the value for this argument in quotes.
-CardType	Alphanumeric	Required	The credit card type. Types are case-insensitive. Valid types are: Visa MasterCard AmericanExpress Discover JCB DinersClub CarteBlanche

Argument	Type	Required/ Optional	Description
-ClientRef	String	Optional	Reference information provided by the customer for tracking purposes.
-Currency	Alpha	Required	The three-character ISO 4217 currency code. Some common codes: USD US dollar CAD Canadian dollar FRF French Franc For the complete list of currency codes, contact the International Standards Organization.
-MerchantRef	Alphanumeric	Optional	Reference information provided by the merchant for tracking purposes. If not specified, "00000000" is used. For FDC, the merchantReference is numeric. Analogous to the **merchantReference** of the **Slip** object (if using LivePayment objects).
-OrdDescFile	String	Optional	The name of the file that contains the order description including type of goods ordered, price, delivery information etc. If not specified, the default is cporddsc.slp.
-PAN	Integer	Required	The personal account number or credit card number.
-PANExpDate	String	Required	The card expiration date (yyyymm).
-PswdFile	String	Optional	The filename of the password file containing the password used to encode/decode the slip. If not specified, the default is ../config/lp-default-pswdfile.

Argument	Type	Required/ Optional	Description
-SlipAmount	Integer	Required	The amount of money. The unit is based on the currency code. For the US dollar the unit is a cent.
-SlipExpDate	String	Optional	The slip expiration date (mmyy). After this date, the merchant stops using the slip.
-SlipFile	String	Optional	The filename of the slip. If not specified, the default is cpslip.slp.

Output This command outputs a “slip created” message if the slip was created successfully.

Example The following example shows an example of the command followed by an example of the output:

```
cpcmd -command createslip -currency USD -amount 1295 -merchantref 1234
-cardtype MasterCard -pan 5200000000000007 -panexpdate 199712
-orddescfile ord.dsc -billstreet "234 First Street" -billzip "94043"
```

```
Slip created.
```

This example creates a slip for an amount of $12.95 in US dollars for a MasterCard card with a card number 5200000000000007 that expires on 12/97. Since no slip file is specified, the slip is in the default file cpslip.slp. Order description information is in the file ord.dsc. The billing street address and zip code are 234 First street, and 94043.

Credit

The **Credit** command sends a credit transaction to the acquirer for all or part of the amount originally captured.

Syntax **cpcmd -Command Credit -MerNum** *merchant_number* **-TermNum** *terminal_number* **-Amount** *amount* **-TranxId** *transaction_id* [**-SlipFile** *slip_file*] [**-MerchantRef** *merchant_reference*] **-BatchNumber** *batch_number* [**-PswdFile** *password_file*]

Arguments The following arguments are available for the **Credit** command:

Argument	Type	Required/ Optional	Description
-Amount	Integer	Required	The amount of money. The unit is based on the currency code. For the US dollar the unit is a cent.
-BatchNumber	Integer	Required	The batch number generated by the **GetCurrentBatch** command.
-MerchantRef	Alphanumeric	Optional	Reference information provided by the merchant for tracking purposes. If not specified, the default is the **MerchantRef** from the slip. Analogous to the **merchantReference** of the **PayEvent** object (if using LivePayment objects).
-MerNum	Alphanumeric	Required	The merchant number, which is provided by the acquirer.
-PswdFile	String	Optional	The filename of the password file containing the password used to encode/decode the slip. If not specified, the default is ../config/lp-default-pswdfile.
-SlipFile	String	Optional	The filename of the slip. If not specified, the default is cpslip.slp.
-TermNum	Alphanumeric	Required	The terminal number, which is provided by the acquirer.
-TranxId	Integer	Required	The transaction ID. It must be unique within the batch.

Output If successful, outputs the amount credited.

Example

```
cpcmd -command credit -termnum 0003277999 -mernum 0001250999 -amount
1295 -tranxid 99913 -batchnumber 00010

credited USD1295
```

This example credits $12.95 in US dollars. It uses the default slip file and password file. It uses the merchant reference from the slip.

GetCurrentBatch

GetCurrentBatch gives the batch number of the current batch. An open batch collects payment information to forward through the Gateway in batch mode.

Syntax **cpcmd -Command GetCurrentBatch -MerNum** *merchant_number* **-TermNum** *terminal_number*

Arguments The following arguments are available for the **GetCurrentBatch** command:

Argument	Type	Required/ Optional	Description
-MerNum	Alphanumeric	Required	The merchant number, which is provided by the acquirer.
-TermNum	Alphanumeric	Required	The terminal number, which is provided by the acquirer.

Output Outputs the current batch number.

Example The following example shows an example of the command followed by an example of the output:

```
cpcmd -command getcurrentbatch -termnum 00003277999 -mernum 00002650999
Batch Number: 00157
```

This example opens a batch for a merchant with a merchant number of 00002650999 and a terminal number of 00003277999. It returns the batch number 00157.

SettleBatch

SettleBatch settles a batch. A batch collects payment information in batch mode.

Some of the arguments for **SettleBatch** are totals of sales and amounts that you have kept track of separately. **SettleBatch** compares the totals in your records with the totals in the batch.

Syntax

cpcmd -Command SettleBatch -MerNum *merchant_number*
-TermNum *terminal_number* **-Currency** *currency*
-MerchantRef *merchant_reference* **-BatchNumber** *batch_number*
[**-TSalesAmt** *total_sales_amount*] [**-TCreditAmt** *total_credit_amount*]
[**-TSalesCount** *total_sales_count*] [**-TCreditCount** *total_credit_count*]

Arguments

The following arguments are available for the **SettleBatch** command:

Argument	Type	Required/ Optional	Description
-BatchNumber	Integer	Required	The batch number generated by the **GetCurrentBatch** command.
-Currency	Alpha	Required	The three-character ISO 4217 currency code. Some common codes: USD US dollar CAD Canadian dollar FRF French Franc For the complete list of currency codes, contact the International Standards Organization.
-MerchantRef	Alphanumeric	Required	Reference information provided by the merchant for tracking purposes. Analogous to the **merchantReference** of the **PayEvent** object (if using LivePayment objects).
-MerNum	Alphanumeric	Required	The merchant number, which is provided by the acquirer.
-TCreditAmt	Integer	Optional	The total credit amount. If not specified, this argument defaults to 0.
-TCreditCount	Integer	Optional	The total number of credits. If not specified, this argument defaults to 0.

Argument	Type	Required/ Optional	Description
-TermNum	Alphanumeric	Required	The terminal number, which is provided by the acquirer.
-TSalesAmt	Integer	Optional	The total sales amount. If not specified, this argument defaults to 0.
-TSalesCount	Integer	Optional	The total number of sales. If not specified, this argument defaults to 0.

Output If successful, outputs a message that the batch has been closed.

Example

```
cpcmd -command settlebatch -mernum 00002650999 -termnum 00003277999
-currency USD -merchantref 1234 -batchnumber 0023 -tsalesamt 55543
-tsalescount 10
```

```
batch 0023 closed
```

This example shows a command to settle the batch numbered 0023. The merchant reference information is 1234. The batch's total sales are $555.43 in US dollars and ten sales make up the batch. Because the **TCreditAmt** and **TCreditCount** are not specified, they default to zero. This batch does not contain any credits.

5

Appendices

- **Troubleshooting LivePayment**
- **Netscape-supported bank card acquirers**

Appendix

A

Troubleshooting LivePayment

This chapter contains information on troubleshooting Netscape LivePayment and the card processor.

This chapter contains the following sections:

- Troubleshooting overview
- Resolving card processor error messages
- Verifying the configuration
- Testing the gateway connection
- Using traceroute and telnet
- Checking the card processor log file
- Technical support

Troubleshooting overview

For bank card processing, the card processor must be running and you must have a current batch. The batch should be closed at the end of a transaction period (for example, at the close of the business day). If you run into trouble, follow these steps:

1. Examine and resolve card processor and connection error messages.
2. Verify the configuration.
3. Check the card processor's connection to the Gateway.
4. Use **traceroute** to identify any Internet connectivity problems.
5. Use **telnet** to identify any Internet connectivity problems.
6. Check the card processor log file.

Resolving card processor error messages

If there is no error message (your machine just hangs), you may have an Internet connection problem.

1. Check all hardware connections and make sure that everything is plugged in securely.
2. Use **traceroute** (and then **telnet**) to the Gateway to see if you have Internet connectivity. Also try to ping some other Internet host—pick your favorite Internet site.
3. If none of this works, your Internet connection is probably down. Contact your ISP. If you do have Internet connectivity, but your Netscape product is still hanging, contact Technical Support.

Refer to the following table for the cause of errors and what action to take. Card processor *connection* error messages are at the end of this table on page 240. Some of the messages contain a **%0** or **%1**. These are variables that will be replaced with a literal string in the error message you see. For example, in the message

"Missing parameter: **%0**"

the **%0** may be "Terminal ID number," and that is what you'd see in the error message:

"Missing parameter: Terminal ID number"

Most of the error messages in this table begin with **cardtxerr**. For simplicity, the messages are listed in numerical order with the **cardtxerr** prefix dropped:

Message	Cause	Action
1000, "Encountered bank card transaction error: %0"	Generic bank card transaction error	Look at additional error message info.
1002, "Bank card processing internal error: %0"	Generic bank card internal error	Contact Technical Support.
1004, "Bank card processing internal error: %0 [%1]"	Generic bank card internal error	Contact Technical Support.
1006, "Missing parameter: %0"	Missing parameter	Check your configuration and permissions.
1008, "Parameter value for %0 is not a valid number"	Parameter value is not a valid number	Check your configuration and permissions.
1010, "Daemon %0 already running"	Daemon already running	Shut down existing daemon first before restarting.
1012, "Batch does not have correct data"	Batch does not have correct data	Check your configuration and permissions. Make sure you have input valid data and that all parameters in your configuration are correct. Make sure the card processor is running.
1014, "Slip does not have correct data"	Slip does not have correct data	Check your configuration and permissions.
1016, "Payment does not have correct data"	Payment does not have correct data	Check your configuration and permissions.
1018, "Payment Event does not have correct data"	Payment Event does not have correct data	Check your configuration and permissions.

Message	Cause	Action
1020, "Cannot allocate memory to create '%0' object"	new() failed	Check your configuration and permissions. Make sure your system has sufficient resources (memory, swap space, etc.) and is not overloaded.
1022, "Invalid processor: %0"	Invalid processor	Check your configuration and permissions.
1024, "Term does not have correct data"	Term does not have correct data	Check your configuration and permissions.
1026, "Merchant does not have correct data"	Merchant does not have correct data	Check your configuration and permissions.
1028, "Cannot open file %0 for reading: %1"	Cannot open file for reading	Make sure file exists and is readable.
1030, "Cannot open ACL file %0 for reading"	fopen could not open ACL file for reading	Verify that the ACL file exists and is readable by user.
1032, "A line in the ACL file is too long:"	A line is terminated by newline or EOF, and should be less than 2K bytes	Verify that all the lines in the ACL file are correct.
1034, "Cannot certify because CA issuer is invalid"	The CA issuer name is not in the ACL file	Verify that all the lines in the ACL file are correct, and compare this CA issuer to those specified in the ACL file. This CA issuer name may also be badly formatted.
1036, "Cannot certify because serial number is invalid"	The serial number is not in the set for the specified CA issuer	Verify that all the lines in the ACL file are correct, and compare this serial number to those in the ACL file. The serial number may also be badly formatted.
1038, "Merchant ID not in the set"	Merchant ID not in ACL file	Verify that all the lines in the ACL file are correct. The merchant ID may also be badly formatted.

Message	Cause	Action
1040, "There is no merchant ID for this set"	There is no merchant ID set for the specified CA issuer and serial number	Verify that all the lines in the ACL file are correct. The serial number may also be badly formatted.
1042, "The ACL file is not in proper format"	A line is terminated by newline or EOF, and should be less than 2K bytes	Check the ACL file. Either a ']' or a delimiter is missing.
1500, "Card processor is not online"	Card processor is not online	Check your configuration and permissions.
1502, "Card processor does not support batch"	Card processor does not support batch	Check your configuration and permissions.
1504, "No batch number"	No batch number	Make sure you get a batch number.
1506, "Information for the given batch object is not available"	Information for the given batch object is not available	Make sure the batch object is supplied with the information.
1508, "Batch response error: %0"	Batch response error	Contact Technical Support.
1510, "Invalid card type: %0"	Invalid card type	Input a valid card type. The values should be: Visa: 012 Mastercard: 112 Amex: 212 Diners Club/CarteBlanche: 312 Discover: 512 JCB: 914
1512, "Invalid Slip Purchase Information Encryption Algorithm ID."	Invalid Slip PI encryption algorithm ID	Make sure you use the correct slip.
1514, "Batch not in open state"	Batch not in open state	Verify that the batch is in open state.
1534, "Invalid Card Number: %0"	Invalid card number	Input a valid card number, with no dashes or spaces. See “Checking for valid bank card numbers” on page 241 at the end of this table.
1540, "Invalid Merchant Reference: %0"	Invalid Merchant Reference	Input a valid Merchant Reference.

Message	Cause	Action
1542, "Invalid Billing Street: %0"	Invalid Billing Street	Check that the bank card billing address and zip code are valid. The address should agree with the address on the bank card billing statements.
1544, "Invalid Billing Zipcode: %0"	Invalid Billing Zipcode	Check that the bank card billing address and zip code are valid. The address should agree with the address on the bank card billing statements.
1546, "Referral - Call Voice Center"	A variety of causes may produce this message, including an expired card.	Call your acquirer's voice center to get approval.
1548, "Declined"	Declined	Check that the card is valid, and that it is not over the credit limit.
1550, "Card Expired"	Card Expired	Use a valid, unexpired card and check that the entered dates are correct.
1552, "Invalid Master Card Summary Invoice: %0"	Invalid Master Card Summary Invoice	Input a valid Merchant Reference.
1554, "Invalid American Express Card Summary Invoice: %0	Invalid American Express Card Summary Invoice	Input a valid Merchant Reference.
1556, "Invalid Diner's Club Card Summary Invoice: %0"	Invalid Diner's Club Card Summary Invoice	Input a valid Merchant Reference.
1558, "Invalid Discover Card Summary Invoice: %0"	Invalid Discover Card Summary Invoice	Input a valid Merchant Reference.
1560, "Invalid Authorization Code: %0"	Invalid Authorization Code	Input a valid Authorization Code.
1562, "Invalid Payment Service Data: %0"	Invalid Payment Service Data	Input valid Payment Service Data.
1564, "Close batch reports out of balance condition"	Close batch reports out of balance condition	Double-check the total amount and count of sales and credit.
1566, "Invalid Visa Card Summary Invoice: %0"	Invalid Visa Card Summary Invoice	Input a valid Merchant Reference.
1568, "Cannot open control file for reading: %0"	Cannot open control file for reading	Make sure the card processor is running and the control file exists and is readable.

Message	Cause	Action
1570, "Cannot open FIFO file for writing: %0"	Cannot open FIFO file for writing	Make sure file exists and is writable.
1572, "Write to FIFO file failed: %0"	Write to FIFO file failed	Make sure file exists and is writable.
1574, "Cannot get message queue ID: %0"	Write to FIFO file failed	Check your configuration and permissions. Make sure the message queue is in good shape.
1576, "Read from message queue interrupted: %0"	Read from message queue interrupted	Determine why the application was interrupted.
1578, "Read from message queue failed: %0"	Read from message queue failed	Make sure the message queue exists.
1580, "Response timeout"	Response timeout	Contact acquirer Technical Support and make sure the card gateway is running. For average response times, see “Response time from the acquirer” on page 242.
2000, "Database query returns no result"	Data not found	Check your configuration and permissions.
2002, "Database value for %0 column is unexpectedly null"	Unexpected null DB result	Check your configuration and permissions.
2004, "Invalid batch state: %0"	bat_status column value	Check your configuration and permissions.
2006, "This merchant and card processor combination has too many batches in open/closing state. [Merchant primary key %0] [Card Processor primary key %1]"	Too many rows returned from DB	Check your configuration and permissions.
2008, "Database server error"	RWDBStatus::serverError	Look at additional error message.
2010, "Database server message"	RWDBStatus::serverMessage	Look at additional error messages.
2012, "Database library error"	RWDBStatus::vendorLib	Look at additional error messages.
2014, "Database error"	other RWDBStatus error	Look at additional error messages.
2016, "Already has an open or closing batch: %0"	You can have only one open or closing batch	Make sure that only one batch is open or closing at a time.

Message	Cause	Action
2018, "Too many rows returned from DB from %0 table"	DB returned too many rows	Check integrity of your DB.
3000, "Invalid Term parameter provided"	A null or invalid term parameter was passed in	Pass in the correct term parameter.
3100, "Term ID: %0, Term num: %1 does not belong to any Merchant"	Cannot locate the Merchant that owns this Term	The terminal number is not correct. Check that you correctly entered the terminal number you received from the acquirer.
3102, "Merchant %0 is not associated with Processor %1"	Merchant has not established the relationship with the Processor	Your merchant number is not valid for your acquirer. Check that you correctly entered the merchant number received from your acquirer.
3200, "Last batch for Term ID %0, Term num %1 is not closed"	Can open a batch for the terminal only when the last batch has been closed	Properly close the batch for the terminal before opening a batch.
3300, "Persistence error [%0] [%1]: [%2]"	Persistence runtime error	Contact Technical Support.
3500, "Invalid Term parameter provided"	A null or invalid Term parameter was passed in	Pass in the correct Term parameter.
3502, "Invalid Batch parameter provided"	A null or invalid Batch parameter was passed in	Pass in the correct Batch parameter.
3504, "Invalid Merchant parameter provided"	A null or invalid Merchant parameter was passed in	Pass in the correct Merchant parameter.
3506, "Invalid Processor parameter provided"	A null or invalid Processor parameter was passed in	Pass in the correct Processor parameter.
3508, "Invalid PayEvent parameter provided"	A null or invalid PayEvent parameter was passed in	Pass in the correct PayEvent parameter.
3510, "Invalid Slip parameter provided"	A null or invalid Slip parameter is passed in	Pass in the correct Slip parameter.
3512, "Order description from Slip and Merchant don't match"	Order description from Slip and Merchant don’t match	Check the steps and content of order description generation.
3514, "Cannot open the configuration file, error %0"	Cannot open the configuration file	Check the location and protection of the configuration file.

Message	Cause	Action
3516, "The data of object %0 is not OK/complete for operation %1"	The data of object is not OK/complete for the operation	Make sure the object is in good status and the relevant properties for the operation have been set properly.
3518, "Encrypted password file is not provided"	Encrypted password file is not provided	Make sure the encrypted password file is provided.
3520, "Invalid date format in property %0.%1"	Invalid date format in the object property	Make sure the date format is in YYYYMM.
3522, "Property %1 in object %0 is not OK/complete for operation %2"	The property in object is not OK/complete for the operation	Make sure the relevant properties in the object for the operation have been set properly.
3524, "Amount %0 in object %1 exceeds the amount in object Slip for operation %2"	The amount in object exceeds the amount in object Slip for the operation	Make sure the amount in the object does not exceed the amount in the object Slip.
4000, "Invalid argument: %0"	An invalid argument	Use the correct argument.
4002, "Duplicate argument: %0"	Duplicate argument	Use the correct argument.
4004, "Missing argument: %0"	Missing argument	Use the correct argument.
4006, "Invalid argument value: %0"	An invalid argument value	Use the correct argument value.
4008, "Missing value for argument: %0"	Missing argument value	Supply the argument value.
4010, "Cannot obtain parameter: %0 [%1]"	Missing parameter	Check your configuration and permissions.
4100, "Invalid number of arguments passed in method %0.%1"	Invalid number of arguments	Use the correct number of arguments.
4102, "No arguments allowed in method %0.%1"	No arguments allowed in this method	Do not pass in arguments to this method.
4104, "Mismatched argument passed in method %0.%1"	Provided argument(s) not matched	Use the correct argument(s).
4120, "Object %0 is not constructed properly"	Object is not constructed properly	Provide proper argument(s) to construct the object.
4122, "Property %0.%1 cannot be set"	This property cannot be set	This property cannot be set.
4124, "Invalid number [%0] is assigned to property %1.%2"	Invalid number is assigned to property	Use the correct number.

Message	Cause	Action
4126, "Missing value in property %1.%2"	Missing value in the property	A value is missing in the property.
4140, "Method %0.%1 failed"	Method failed	Check the arguments and the status of the object.
4190, "LivePayment software has expired"	LivePayment software has expired	Upgrade LivePayment to a newer version or contact Technical Support for details.
5000, "Card Processor Error %0 -- Invalid transaction code: %1"	Invalid transaction code	Input a valid transaction code.
5002, "Card Processor Error %0 -- Terminal ID not set up to capture this card type"	Terminal ID is not set up to capture this card type	Input a valid Terminal ID/card type combination or contact acquirer Technical Support to accept this card type.
5004, "Card Processor Error %0 -- Terminal ID not set up to authorize this card type	Terminal ID not set up to authorize this card type	Input a valid Terminal ID/card type combination or contact acquirer Technical Support to accept this card type.
5006, "Card Processor Error %0 -- Invalid card expiration date"	Invalid card expiration date	Input a valid expiration date in the correct format, using an unexpired card.
5008, "Card Processor Error %0 -- Invalid Process Code, Authorization Type or Card Type"	Invalid Process Code, Authorization Type or Card Type	Contact Technical Support.
5010, "Card Processor Error %0 -- Invalid Transaction or Other Dollar Amount: %1"	Input a valid Transaction or Other Dollar Amount	Input a valid Transaction Amount. If FDC is your acquirer: For auth/capture, the max amt. is 7 digits. For close batch, the max amt. is 9 digits (including the 2 decimal digits for cents, but excluding the decimal itself).
5012, "Card Processor Error %0 -- Invalid Entry Mode"	Invalid Entry Mode	Contact Technical Support.
5014, "Card Processor Error %0 -- Invalid Card Present Flag"	Invalid Card Present Flag	Contact Technical Support.
5016, "Card Processor Error %0 -- Invalid Customer Present Flag"	Invalid Customer Present Flag	Contact Technical Support.

Message	Cause	Action
5018, "Card Processor Error %0 -- Invalid Transaction Count Value"	Invalid Transaction Count Value	Contact Technical Support.
5020, "Card Processor Error %0 -- Invalid terminal type"	Invalid terminal type	Contact Technical Support.
5022, "Card Processor Error %0 -- Invalid terminal capability"	Invalid terminal capability	Contact Technical Support.
5024, "Card Processor Error %0 -- Invalid source ID"	Invalid source ID	Check your configuration and permissions.
5026, "Card Processor Error %0 -- Invalid batch number"	Invalid batch number	Input a valid batch number. It should be numeric and increase by one. If your next batch is a random number, then you're using the dummy test.
5028, "Card Processor Error %0 -- Invalid mag stripe data"	Invalid mag stripe data	Contact Technical Support.
5030, "Card Processor Error %0 -- Invalid Merchant Reference"	Invalid Merchant Reference	Input a valid Merchant Reference.
5032, "Card Processor Error %0 -- Invalid transaction date or time"	Invalid transaction date or time	Contact Technical Support.
5034, "Card Processor Error %0 -- Invalid card processor merchant number: %1"	Invalid merchant number in card processor DB	Input valid merchant number.
5036, "Card Processor Error %0 -- File access error in card processor DB"	Encountered file access error in card processor DB	Contact acquirer Technical Support.
5038, "Card Processor Error %0 -- Terminal flagged as inactive in card processor DB"	Terminal flagged as inactive in card processor DB	Contact acquirer Technical Support.
5040, "Card Processor Error %0 -- Invalid Merchant/Terminal ID combination"	Invalid Merchant/Terminal ID combination	Input a valid Merchant/Terminal ID. Check the Merchant/Terminal IDs and restart the server. Confirm IDs with acquirer.
5042, "Card Processor Error %0 -- Unrecoverable card processor DB error"	Card processor encountered unrecoverable DB error from an authorization process	Contact acquirer Technical Support.
5044, "Card Processor Error %0 -- DB access lock encountered"	Card processor DB access lock encountered	Contact Technical Support.

Message	Cause	Action
5046, "Card Processor Error %0 -- DB error in summary process"	Encountered card processor DB error in summary process	Contact Technical Support.
5048, "Card Processor Error %0 -- Transaction ID invalid, incorrect, or out of sequence"	You have given an invalid, incorrect, or out of sequence Transaction ID.	Input a valid Transaction ID. .
5050, "Card Processor Error %0 -- Terminal ID not usable"	Terminal ID not usable	Input a valid Terminal ID.
5052, "Card Processor Error %0 -- Terminal ID not set up: %1"	Terminal ID not set up	Input a valid Terminal ID.
5054, "Card Processor Error %0 -- Capture transaction for batch where earlier Batch ID still open"	Capture transaction for batch where earlier Batch ID still open	Contact acquirer Technical Support.
5056, "Card Processor Error %0 -- Invalid account number found by authorization process"	Invalid account number found by authorization process	Input a valid card number.
5058, "Card Processor Error %0 -- Invalid capture data found in batch process (trans level)"	Invalid capture data found in batch process (trans level)	Input valid capture data.
5060, "Card Processor Error %0 -- Invalid capture data found in batch process (batch level)"	Invalid capture data found in batch process (batch level)	Input valid capture data.
5062, "Card Processor Error %0 -- General system error"	General system error	Contact acquirer Technical Support.
5064, "Card Processor Error %0 -- Invalid Payment Service Data"	Invalid Payment Service Data	Input valid Payment Service Data.
5066, "Card Processor Error %0 -- Unknown processor error"	Unknown card processor error	Contact acquirer Technical Support.
Card processor connection errors		
Error:-1. Cannot establish secure connection to the Gateway.	Socket Error.	Make sure your machine is not running out of resources and networking is functioning properly.

Message	Cause	Action
Error:-2. Cannot establish secure connection to the Gateway.	Invalid Hostname for Gateway. You may also get this error if the Gateway certificate has expired.	1. Try again. 2. Verify the card processor host name. 3. Try traceroute to your Gateway. If successful, try the card processor again. 4. Try telnet to your Gateway. If successful, try the card processor again. 5. If none of the above works, your Internet connection is probably down. Contact your ISP.
Error:-3. Cannot establish secure connection to the Gateway.	SSL Connect Failed.	1. Try again. 2. Verify the host port number in the LivePayment configuration. 3. Try traceroute to your Gateway. If successful, try the card processor again. 4. Try telnet to your Gateway with the port number. If successful, try the card processor again. 5. If none of the above works, contact Technical Support to make sure your IP address is authorized and to ask for a valid port number.
Error:-4. Cannot establish secure connection to the Gateway.	Security Database Error.	Check your configuration to see that all parameters are set correctly, especially the parameters related to certificate and key pair files.

Table notes

Checking for valid bank card numbers

The following table lists valid formats for bank card numbers at the time of this writing:

Table 10.1 Valid bank card number formats

Card name	Valid format
Visa	First digit must be 4 and the length must be 13 or 16 digits.
AMex	First digit must be 3 and second digit a 4 or 7. The length must be 15 digits.
Diners Club & CarteBlanche	First digit must be 3 and second digit 0, 6, or 8. The length must be 14 digits.
Discover	First four digits must be 6011 and the length must be 16 digits.
JCB	First four digits must be 3088, 3096, 3112, 3158, 3337, or 3528. The length must be 16 digits.

Response time from the acquirer

Transactions across the Internet generally have the following response times when you are running in production mode:

Table 10.2 Estimated response times

Transaction	Estimated Time
Authorize	Less than 10 seconds
Capture	Less than 10 seconds
Get Current Batch	Approximately 1 second
Settle Batch	Approximately 1 second

If no response is received in 50 seconds, the system will timeout.

Verifying the configuration

Some errors are the result of the LivePayment and card processor configuration. To check your configuration, use the configuration forms. For more information, see Chapter 2, “Setting up Netscape LivePayment”.

Testing the gateway connection

To determine whether you can communicate with your Gateway, use the **Test Connection** button on the Card Processor Administration page. For more information, see Chapter 2, “Setting up Netscape LivePayment”.

Using traceroute and telnet

Currently, many applications will only use the first IP address in a multiaddress list (although there are a number of common TCP/IP application programs, like **telnet**, which when unable to connect to the first address in a list, will automatically attempt to connect with other addresses on the list).

You may use the Unix **host** or **nslookup** commands to display the IP address list (rather than just the first address in a list):

```
host ccgw.card.net
```

or

```
nslookup
server hostname
```

Between LivePayment and the acquirer gateways are various Internet hosts, including packet-filtering firewalls. These firewalls maintain IP addresses of machines that are authorized to access the Gateway. If a machine is not on the authorization list, a **telnet** command from that machine will not be allowed to connect to the LivePayment service port (even when the **traceroute** command indicates general Internet connectivity).

In rare situations, where **traceroute** indicates connectivity and a server is known to be authorized, but the **telnet** command is not able to connect, the card processor service at that particular gateway may be temporarily unavailable.

1. Use the **traceroute** command to determine Internet connectivity to a specific gateway.
2. Use the **telnet** command (specifying the card processor service port) to determine access authorization and service availability at a specific gateway.

Using traceroute

Note The **traceroute** utility is available only on some Unix systems. For example, Solaris™ doesn't ship with this utility. It is widely available from various sources on the Internet, however.

Use **traceroute** to determine Internet connectivity to a specific gateway (**traceroute** uses only the first address in the case of a multi-address list):

1. traceroute ccgw1.card.net
2. traceroute ccgw2.card.net

} ✔Check to see that both gateways are up.

3. traceroute ccgw.card.net — **traceroute** will trace the route of one of the two

Here is an example of a successful **traceroute** connection:

Sample traceroute output

```
% traceroute ccgw.card.net
traceroute to ccgw.card.net (204.254.78.2), 30 hops max, 40 byte packets
1 unknown.netscape.com (198.93.93.2) 18.64 ms 7.103 ms 4.165 ms
2 border2-hssi3-0.SanFrancisco.mci.net (204.70.33.9) 3.258 ms 19.778 ms 3.937 ms
3 borderx2-fddi0-0.SanFrancisco.mci.net (204.70.3.164) 7.772 ms 3.73 ms 4.413 ms
4 fix-west-nap.SanFrancisco.mci.net (204.70.158.118) 25.168 ms 5.252 ms 5.651 ms
5 san-jose5.ca.alter.net (198.32.136.42) 223.676 ms 364.317 ms 56.731 ms
6 San-Jose7.CA.ALTER.NET (137.39.27.3) 14.772 ms 13.191 ms 18.256 ms
7 Palo-Alto1.CA.ALTER.NET (137.39.29.3) 31.261 ms 12.756 ms 11.697 ms
8 card-gw.CA.ALTER.NET (137.39.246.58) 20.832 ms 19.988 ms 15.63 ms
9 fw2.card.net (204.254.78.2) 14.91 ms 15.596 ms 19.142 ms
```

If your packet is detained, **traceroute** shows where the problem occurred. If **traceroute** finds a problem with your packet's transmission through the Internet, the problem is probably beyond your control. Wait until the problem with the errant host has cleared, or notify the responsible system administrators or your Internet Service Provider (ISP). Refer to the **traceroute** man page for detailed information.

Using telnet

There are other failure modes besides Internet connectivity. Your IP address might not be in the authorization list in the packet filtering router. There is currently no way of testing for that with **traceroute**.

To determine authorization, as well as the ability to establish a TCP session between the merchant machine and the firewall, use **telnet**:

`telnet` *`host port`* (host and port provided by acquirer)

For example:

1. telnet ccgw1.card.net 999
2. telnet ccgw2.card.net 999
3. telnet ccgw.card.net 999

} ✔Check to see that both gateways are up.

telnet will randomly connect to the first of the two available addresses

Sample telnet output

Successful connection

```
% telnet ccgw.card.net 999
Trying 165.90.142.2 ...
Connected to ccgw.card.net.
Escape character is '^]'.
```

Unsuccessful connection

```
% telnet ccgw.card.net 999
Trying 204.254.78.2...
telnet: connect to address 204.254.78.2: Connection timed out
```

Checking the card processor log file

Check the card processor log file to determine where the link may have broken. You designate the log file when you configure the card processor parameters. For more information, see “Configuring the card processor parameters” on page 33.

The following sample log file output shows you a successful transaction. An example of a broken link is at the end of the following sample log. You can also tell how long it takes to perform transactions. In the following example, one second elapsed between the time the request was read and a response was sent back to the client.

Sample card processor log file:

process id (pid) date/time message

Certificate information

```
(4003)1996.01.24/22:52:01: Peer/Gateway's certificate:

(4003)1996.01.24/22:52:01: Version: 00
Serial Number: 02:7A:00:00:FE
Issuer:  C=US, O=RSA Data Security, Inc., OU=Secure Server Certification
Authority
Subject: C=US, ST=California, L=Mountain View, O=Netscape
Communications,
OU=NOT A VALID CERTIFICATE. DO NOT TRUST. DO NOT XFER SECURE DATA.,
CN=notavalidhost.mcom.com

(4003)1996.01.24/22:52:01:
```

Read request message from the client

```
(4003)1996.01.24/22:52:05: CCPD:read:fifo check input

(4003)1996.01.24/22:52:05: CCPD:check_input: ThreadID: 4012 CmdType: 0
ReqLen 42 RespLen 60 Version 0 CDPType 0

(4003)1996.01.24/22:52:05: CCPDProcessFifo: Msg 0xc6160, ThreadID 4012
Type 0,tx_code 006, term_id 00003277845
```

Read response message from the Gateway

```
(4003)1996.01.24/22:52:06: CCPD:read:ssl check reply

(4003)1996.01.24/22:52:06: CCPD:check_input: ThreadID: 4012 CmdType: 0
ReqLen 42 RespLen 60 Version 0 CDPType 0

(4003)1996.01.24/22:52:06: CCPDSslRead: Msg 0xc6160, ThreadID 4012
```

Send response back to the client

```
(4003)1996.01.24/22:52:06: RespMsg: tx_code 007, term_id 00003277845,
tx_id, resp A, err_code 00

(4003)1996.01.24/22:52:06: CCPDRespondFdc: Msg 0xc6160, ThreadID 4012
```

Sample showing broken connection

The following shows what you can expect to see if the Internet SSL is down. The card processor will keep trying until successful. If this doesn't happen within 15 minutes or so, try checking for Internet connectivity using **telnet** or **traceroute**. Contact your ISP if your connection is down. Contact Technical Support if you have Internet connectivity but still get this message:

SSL connection is down. Keeps trying

```
(947)1995.11.02/19:29:50: CCPD:NetRead: SSL connection disconnected
```

SSL connection reestablished

```
(947)1995.11.02/19:32:44: CCPD:: SSL re-connection successful
```

Technical support

If you're unable to resolve the problem, call Technical Support at (415) 937-2727. You'll need to provide the following information:

- Customer number
- Product and version numbers
- Description of the problem; if there is an error message, record the exact message and keep it on hand
- Steps to recreate the problem (if possible)

For non-urgent problems, send email to iapps-support@netscape.com. Netscape will respond in one business day.

Appendix

B

Netscape-supported bank card acquirers

This appendix contains information on the following Netscape-supported bank card acquirer:

- First Data Corporation (FDC)

First Data Corporation (FDC)

Netscape Communications has established a relationship with First Data Corporation (FDC), a financial institution that can assist you in locating an acquirer. For additional information about FDC and their Internet card processing service, visit the following Web site:

```
http://www.firstdata.com/ecom.html
```

You can also send e-mail to emerchant@netscape.com or call (415)842-4085.

Establishing a credit card business agreement

In order to run your application in production mode, you must establish a credit card business agreement with a bank card acquirer. After your business agreement is in place, your bank card acquirer should provide you with the following parameters:

- Merchant number
- Terminal number
- Merchant source ID
- Credit card gateway host name
- Credit card gateway port number

This information is required for configuring the LivePayment credit card processor. For configuration information regarding credit card transactions, see Chapter 2, “Setting up Netscape LivePayment”.

For additional information about establishing a merchant account, you can visit the Netscape home page at

```
http://home.netscape.com/eng/LivePayment
```

Obtaining test processing parameters

In order to properly test your LivePayment application as a developer or while waiting for the setup of your merchant account, you must enable communications with a test bank card gateway. Netscape has fielded a test bank card gateway at FDC so that your LivePayment-based application can send financial transactions and receive responses in a test environment.

To communicate with that gateway you need to fill in a brief application form. For more details, see the Netscape home page at:

```
http://home.netscape.com/eng/LivePayment
```

Click **Developer Sign-up** for more details. After filling in the necessary forms, you will receive the needed data parameters. You need to obtain your certificate before filling in the requested form.

When you receive your test values, you can change from operating in loopback mode to operating in test mode. For more information on test mode and loopback mode and changing between them, see “Changing LivePayment operating modes” on page 28.

Getting information about your certificate

As part of the application process with FDC, you need to know the issuer information and the serial number of your certificate. You can find this information by clicking the Examine a Certificate link on the Netscape LivePayment page. For more information, see “Examining a certificate” on page 45.

Certifying your application

Before going live (entering production mode) with your LivePayment application, you must contact FDC to get it certified. Certification assures that your application meets credit card processing standards. You need your application certified under the following circumstances:

- If you create an application from scratch (without using the Starter Application Set).
- If you modify the Starter Application Set (LPStart and LPAdmin) by changing the .js files in the readonly_lib directory.

If you modified the Starter Application Set without modifying the .js files, your application will not need to go through the whole certification process. Netscape or FDC will verify that your .js files have not been modified.

In general, the certification process checks that:

- The data forwarded to the acquirer is the correct data in the correct format.
- The application can correctly handle the information received from the acquirer.
- Standard business practice rules are not violated.

The certification is handled on a one-on-one basis by FDC. For certification, contact FDC.

Index

A

acquirer
  defined **16**
  establish service with **27**
  First Data Corporation **249**
  information provided by **27**, **64**
  Netscape supported **249**
  response time **242**
address verification service result **64**, **178**
administer LivePayment **45**
administer LiveWire **47**
amount property
  PayEvent object **173**
  Slip object **174**
appendMerchantOrderDesc
  method **174**
appendOrderDesc method **175**
application
  certify **251**
authCode property **176**
authorization code **64**
authorize **164**
  business rules **55**
  cpcmd **214**
  information required for **53**
  results **64**
Authorize function **84**, **114**
authorize method **177**
AuthToCapturing function **115**
AVS
  See address verification service result
avsResp property **178**

B

bad method **150**, **178**
bank card number
  valid **241**
batch
  create **53**
  for capture **59**
  for credit **59**
  for settle **59**
  number
    get **58**
    get with cpcmd **223**
  settle **80**, **92**
  state **65**
  transaction order **60**
Batch object **155**, **179**
  properties **155**
batch object **87**
batchNumber property **180**
billingStreet property **181**
billingZip property **181**

C

CalcOrderTotal function **115**
calcScore function **115**
Cancel function **84**, **116**
capture **165**
  business rules **56**
  information required for **54**
  using batch number **59**
  with cpcmd **217**
Capture function **84**, **92**, **116**
capture method **181**

card processor
  defined **16**
  error messages **230**
  log file **245**
  parameters **33**
  start **46**
  stop **47**
cardExpirationDate property **182**
cardNumber property **182**
cardType property **183**
CentsToDollarStr function **116**
certificate
  card processor **38**
  distinguished name **43**
  examine **45**
  file
    card processor parameter **34**
  install **44**
  request **41**
  use server **36**
certificate authority **38**, **42**
certify application **251**
CheckAVS function **117**
clearStatus method **151**, **183**
common gateway interface (CGI)
  defined **16**
common name **43**
confirm function **117**
control file **35**
countActiveParams function **117**
cpcmd utility
  commands **210**
  overview **210**
CreateCreditevent function **118**
CreateManualCreditEvent function **118**
CreatePayevent function **118**
CreateSlip
  cpcmd **219**
credit **166**
  business rules **56**
  information required for **54**
  using batch number **59**
  with cpcmd **221**
credit card processing
  functions **84**
Credit function **84**, **92**, **119**
credit method **184**
creditCount property **184**
cryptography **23**
currency property
  Batch object **185**
  Slip object **185**
cursor method **85**
cursor object **85**

D

database
  design **57**
database object **85**
database schema **109**
database tables **110**
develop
  LivePayment application
    steps **144**
distinguished name **43**
doCredit function **119**
DollarStrtoCents function **119**
Domain Name Service (DNS) **35**
dynSelect function **120**

E

emitFooter function **120**
emitHeader function **120**
encode method **160**, **186**
encryptPasswordFile property **187**

error messages
  card processor **230**
error status methods **150**
eventID property **187**
eventTime property **188**
execute method **85**

F

failOnAVS function **121**
FIFO file **35**
First Data Corporation
  establishing service with **249**
function
  livePaymentVersion **192**
  registerLivePayment **146**, **199**
  registerNativeFunction **146**, **199**

G

gateway
  connection test **46**
  defined **16**
GenerateMerchantReference function **121**
GenerateSlip function **89**, **121**
GetCardType function **122**
GetCurrentBatch function **84**, **122**
getCurrentBatch method **188**
getDER method **189**
GetItemProperties function **123**
GetNextBatchID function **123**
GetNextEventID function **123**
GetNextPurchaseID function **123**
GetNextSlipID function **124**
getStatusCode method **150**, **190**
getStatusMessage method **151**, **190**
getTitleString function **124**
good method **150**, **191**

H

host name **34**

I

idempotent
  defined **68**
  transactions **68**
initMerchantOrderDesc method **191**
isAmericanExpress function **124**
isAmEx function **124**
isAnyCard function **125**
isBlank function **125**
IsCardMatch function **125**
isCarteBlanche function **126**
isCB function **126**
isCC function **126**
isDC function **126**
IsDiners function **126**
isDinersClub function **126**
isDiscover function **126**
isenRoute function **127**
IsExistingPurchaseID function **127**
isJCB function **127**
isMasterCard function **127**
isMastercard function **127**
isMC function **127**
isNum function **128**
isValidDay function **128**
isValidPrice function **128**
isValidPurchaseid function **128**
isValidYear function **129**
isValidZip function **129**
isVISA function **129**
isVisa function **129**
ItemObject function **129**

J

JavaScript
  defined **15**
  embed in HTML **146**

K

key
  private **38**
  public **38**
key pair file
  card processor parameter **34**
  generate **38**
  location **42**
  password **39**, **43**
    change **40**

L

LivePayment objects **86**
livePaymentVersion function **192**
LiveWire
  application manager **48**
  database connectivity tools **21**
  defined **14**
log file **35**
  card processor **245**
loopback mode
  See mode, loopback
LPAuthOnly sample application **167**

M

makeValidMerchantReference function **130**
MatchShipToBill function **130**
Merchant object **87**, **152**, **192**
  properties **153**
merchantNumber property **193**
merchantReference property
  Batch object **194**
  PayEvent object **194**
  Slip object **195**
method
  appendMerchantOrderDesc **174**
  appendOrderDesc **175**
  authorize **177**
  bad **150**, **178**
  capture **181**
  clearStatus **151**, **183**
  credit **184**
  encode **160**, **186**
  error status **150**
  getCurrentBatch **188**
  getDER **189**
  getStatusCode **150**, **190**
  getStatusMessage **151**, **190**
  good **150**, **191**
  initMerchantOrderDesc **191**
  Processor object **154**
  settleBatch **201**
  Slip object **159**
mode
  loopback
    change to test **29**
    defined **28**
    start **29**
  operating, defined **28**
  production
    defined **28**
    start **30**, **31**
  test
    change to production **30**
    defined **28**
    start **29**

N

name property
  Merchant object **195**
  Processor object **196**

O

object
  Batch **155**, **179**
  create **148**
  Merchant **152**, **192**
  PayEvent **162**, **196**
  Processor **154**, **197**
  Slip **157**, **202**
  Terminal **152**, **204**
operating system shell **15**

P

parameters
  card processor
    configure **33**
  default values from **149**, **211**
  LivePayment
    configure **32**
PayEvent object **162**, **196**
PayEvent object properties **162**
payment state **65**
paySvcData property **197**
port number **34**
preparetoSettle function **131**
PrintAVSError function **131**
PrintBatchItem function **131**
PrintError function **132**
PrintFmtError function **132**
PrintReceipt function **132**
PrintReceiptItem function **133**
PrintSettledBatch function **133**
PrintTrans function **133**
PrintTransItem function **134**
private key **38**
Processor object **87**, **154**, **197**
  methods **154**
  properties **154**
project object **86**
property
  amount
    PayEvent **173**
    Slip **174**
  authcode **176**
  avsResp **178**
  Batch object **155**
  batchNumber **180**
  billingStreet **181**
  billingZip **181**
  cardExpirationDate **182**
  cardNumber **182**
  cardType **183**
  creditCount **184**
  currency
    Batch **185**
    Slip **185**
  encryptPasswordFile **187**
  eventID **187**
  eventTime **188**
  Merchant object **153**
  merchantNumber **193**
  merchantReference
    Batch **194**
    PayEvent **194**
    Slip **195**
  name
    Merchant **195**
    Processor **196**
  PayEvent object **162**
  paySvcData **197**
  Processor object **154**
  purchaseRequestTime **198**
  salesCount **200**
  Slip object **157**
  Terminal object **153**
  terminalNumber **205**
  totalCreditAmount **205**
  totalSalesAmount **206**
  use **148**
public key **38**
purchase
  authorize **88**
purchaseRequestTime property **198**

Q

Query function **134**
QueryCleanup function **134**

R

register LivePayment objects **146**
registerLivePayment function **146**, **199**
registerNativeFunction function **146**, **199**
RemoveAlpha function **134**
request object **86**
response time from acquirer **242**

S

SaleObject function **135**
salesCount property **200**
sample application
  LPAuthOnly **167**
SaveCreditevent function **136**
SavePayevent function **136**
server
  administration **16**
  HTTP **15**
settle **166**
  business rules **57**
  information required for **55**
  using batch number **59**
  with cpcmd **223**
SettleBatch function **84**, **92**, **136**
settleBatch method **201**
shell
  operating system **15**
slip
  create **52**
  create with cpcmd **219**
  encode **89**, **160**
  generate **88**
Slip object **87**, **157**, **202**
  methods **159**
  properties **157**
source ID **34**
starter application
  administrate **77**
  batch states **138**
  configure **76**
  directory structure **104**
  functions **114**
  HTML pages **105**
  libraries **108**
  modify **83**
  modifying **96**
  prerequisites **76**
  transaction states **138**
StateList function **137**

T

technical support **247**
telnet **245**
Terminal object **88**, **152**, **204**
  properties **153**
terminalNumber property **205**
totalCreditAmount property **205**
totalSalesAmount property **206**
TraceFile
  cpcmd **213**
TraceLevel
  cpcmd **213**
traceroute **244**
transaction
  authorize **90**
  cancel **78**
  capture **78**, **91**
  credit **92**
transactions
  view **78**

U

URL
  LivePayment **31**
utility
  defined **15**

V

VerifyPurchaseData function **137**
version
  finding LivePayment **152**
viewTransaction function **137**