OneStep Charge^{SM} User's Guide & API Specification

Version 2.2
Updated March 15, 2011

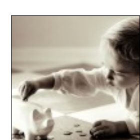 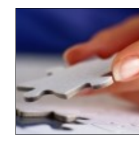

Table of Contents

1.0 Conventions....1
2.0 Introduction....1
3.0 Installation....3
3.1 Installing, Running and Stopping the OneStep Charge Broker (OSCBroker)....3
3.1.1 OSCBroker Command-Line Parameters....5
3.1.2 OSCBroker Single Settlement Module (OPTIONAL)....6
3.2 Installing the OneStep Charge Progress Code....6
4.0 Configuration....7
4.1 OSCConfig Temp-Table....7
4.2 Configuration File....9
5.0 OneStep Charge API Specification....9
5.1 trExec Temp-Table....9
5.2 Credit Card Action (CCA) Values....18
5.3 ACH Action (ACHA) Values....25
5.4 OSC Administrative Action (OSCA) Values....27
5.5 Credit Card Status (CCS) Values....28
5.6 ACH Status (ACHS) Values....30
5.7 Currency Code (OSC_CURR) Values....30
5.8 AVS Result (AVS) Values....32
5.9 API Functions....32
5.10 Multi-Merchant Support....33
5.11 Purchase Card Level II (PC II) Support....34
5.12 Purchase Card Level III (PC III) Support....34
5.13 OneStep Charge Vault....35
5.14 Transaction Settlement....37
5.15 Data Availability Delay....39
6.0 Examples....39
6.1 CCA_AUTHONLY Example....40
6.2 CCA_PURCHASE Example....41
6.2.1 CCA_PURCHASE Example - Purchase Card Level II (PC II)....42

6.2.2 CCA_PURCHASE Example - PC Level III (PC III), No Lines................43
6.2.3 CCA_PURCHASE Example - PC Level III (PC III), Lines.....................44
6.3 CCA_STATUS Example...46
6.4 CCA_CAPTURE Example...46
6.5 CCA_QTRANS Example...47
6.6 OneStep Charge Vault Examples...48
6.6.1 Vault Example: CCA_STORE...48
6.6.2 Vault Example: CCA_STORE (update existing record).........................49
6.6.3 Vault Example: CCA_AUTHONLY (no overridden fields)......................50
6.6.4 Vault Example: CCA_AUTHONLY (overridden fields)...........................51
6.6.5 Vault Example: CCA_QVAULT...51
6.6.6 Vault Example: CCA_UNSTORE...52
6.7 ACHA_PURCHASE Example..53
6.8 Test Credit Card Numbers and ACH Accounts.......................................54
7.0 Diagnostics..55
8.0 Getting Help...55

Copyright © 2003-2011 Tri-8, Incorporated. ALL RIGHTS RESERVED. Unauthorized modification with or without redistribution is strictly prohibited.

Thank you for your interest in OneStep Charge℠, the premier electronic payment processing solution for use within the Progress 4GL. This User's Guide is intended to help you familiarize yourself with OneStep Charge℠ and its operation, and to help you minimize the time necessary to implement OSC at your site and/or with your application.

1.0 Conventions

Throughout the OneStep Charge℠ documentation set you will find the "$OSC" convention used to refer to the installation directory for OneStep Charge℠. This corresponds to the directory into which you have extracted/untarred the OSC (OneStep Charge℠) distribution. For example, if you have untarred your OneStep Charge℠ distribution into /opt on a UNIX machine, $OSC will refer to "/opt/OneStepCharge" for your installation.

When a code snippet or command line is used for illustration or to provide an example, the code or command itself will appear in `this fixed-width font`. For UNIX/DOS command prompts, the prompt itself (configurable on each system, of course) will be represented by "`prompt>`" in this guide.

2.0 Introduction

OneStep Charge℠ is a payment processing engine for the Progress 4GL. Its goal is to enable you to integrate lowest-cost realtime electronic payments into your Progress application with minimal effort and time requirements. The OneStep Charge℠ Applications Programming Interface (API) exists as a defined mechanism whereby you may communicate back and forth with OneStep Charge℠, using OSC as your payment processing backend. Communication with OSC takes place via a standard temp-table named trExec, which is defined in the API include file (tri8osc/oscapi). In general, this communication is as simple as creating one or more trExec records and passing them to the OSC Engine (tri8osc/oscengin.p), then reading those records to determine the results.

OneStep Charge℠ provides the full, robust framework and API for your application in the Progress 4GL, as well as a TCP/IP-based processing node infrastructure whereby transactions are sent to the appropriate processing network, such as Vital, Paymentech, Nova, etc. OSC utilizes a 4GL codebase that imposes a very small footprint in your own application codebase, and the API is written with complete application independence in mind. Therefore, developers can incorporate OneStep Charge℠ functionality into any Progress 4GL Version 9 (or later) program regardless of the user interface (or lack thereof) chosen for the host application.

When OneStep Charge℠ processes a transaction, the transaction data are sent through a triple-redundant, high-performance, encrypted system of processing nodes referred to as the OneStep Charge℠ Engine. The Engine handles high-speed communications with the appropriate processing network (such as Vital) and encrypted storage of essential transaction data elements. A querying infrastructure is placed within the Engine as well, which utilizes a

double-failover (currently) setup. The transaction-processing side of the OSC Engine, complete with triple-failover (currently N=3), is built for 100% uptime, while the querying side (currently double-failover), for CCA/ACHA_STATUS and CCA/ACHA_QTRANS, is only slightly less redundant. Please also refer to the “Data Availability Delay” section in this guide.

The basic architecture and process flow of OneStep Charge℠ is shown in the Figure 1.

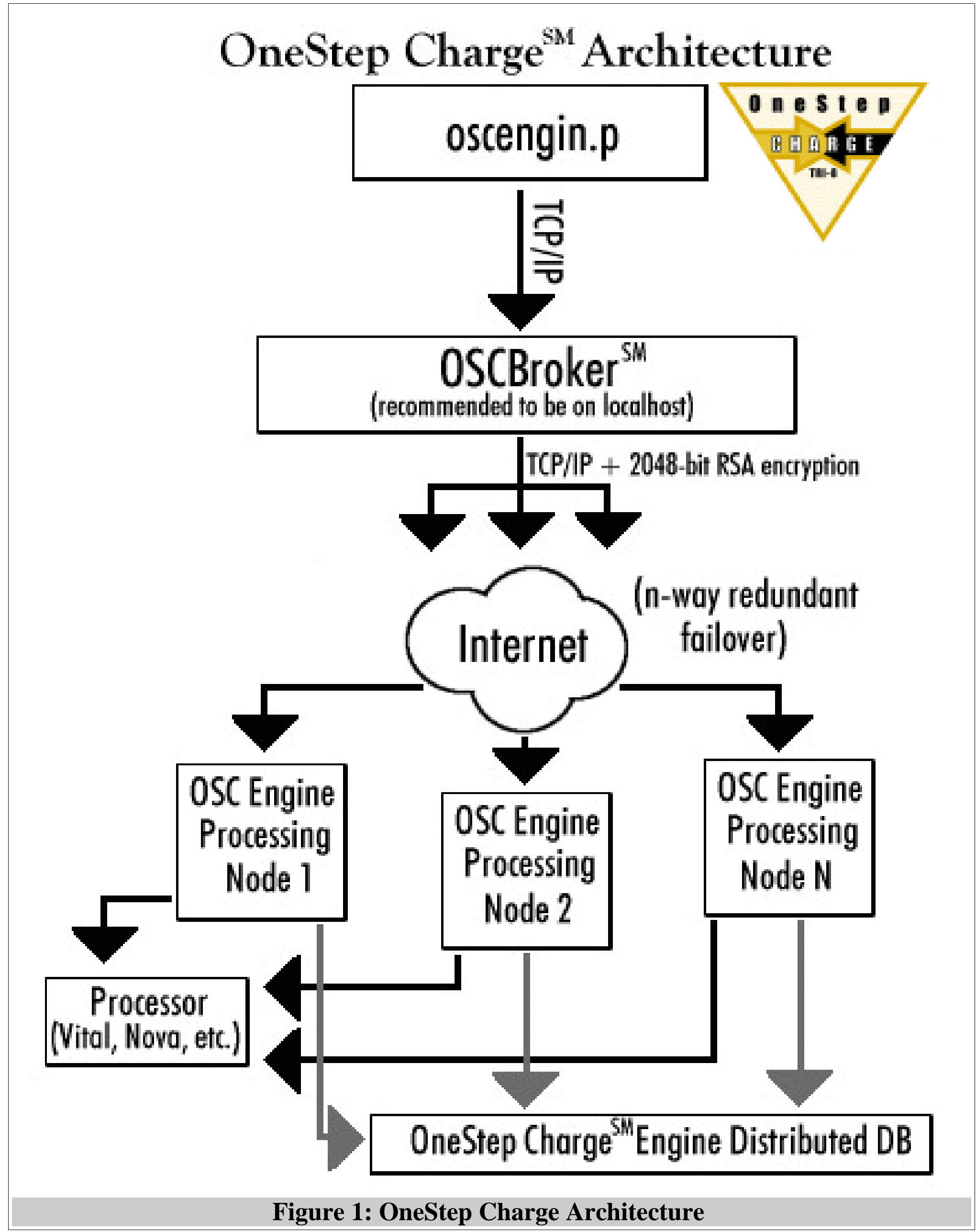

Figure 1: OneStep Charge Architecture

IMPORTANT NOTE: OneStep Charge℠ provides the tools you need for complete electronic payment processing operations within the Progress environment. No database is required and OSC does not store your transactions for you locally. It is always assumed that you will

store the portions of information about transactions that you see fit into your Progress database, in a table of your choice and construction. Once a transaction has either been declined (CCS_DECL, CCS_CALL) or subject to a non-system error (CCS_ERR), or has settled (CCS_CLOSED), the transaction is NOT GUARANTEED to be available for further querying via the OneStep Charge℠ Engine. These "historical" transactions, as opposed to live batches and pre-authorized (CCS_AUTHONLY) transactions, are not kept indefinitely by the payment processing networks nor the OneStep Charge℠ Engine. Transactions are not guaranteed to be available after a period of 120 days from the settlement of the transaction (CCS_CLOSED) or its declination (CCS_DECL). As you will see below, the CCA_QTRANS command is available for reporting/query operations, but this is not intended to provide long-term historical data access.

3.0 Installation

Two main steps make up the installation of OneStep Charge℠:

1. Install and run the OSCBroker℠ (see notes below to skip this step for demo).
2. Unpack and copy the OSC source code into your PROPATH.

3.1 Installing, Running and Stopping the OneStep Charge℠ Broker (OSCBroker℠)

The OSCBroker℠ must be running at all times in order for OneStep Charge℠ to operate. The OSCBroker℠ facilitates connections among the OneStep Charge℠ Engine processing nodes, providing triple redundancy and 2048-bit RSA encryption for every transaction. It is a Java-based utility and requires the Java Runtime Environment (JRE or JDK/SDK) version 1.4.0 or later.

> **PLEASE NOTE**
> You may skip the OSCBroker℠ Java requirements, installation and execution when you first evaluate and test OneStep Charge℠. The default OneStep Charge℠ configuration (in $OSC/tri8osc/oscapi) directs OneStep Charge℠ to connect to the publicly-available DEMO OSCBroker℠ at OSCBrokerDemo.OneStepCharge.com. **THIS IS FOR DEMONSTRATION AND TESTING PURPOSES ONLY AND SHOULD NEVER BE DONE IN A LIVE SCENARIO**. You must always deploy a local OSCBroker℠ before running live transactions or sending **ANY** live credit card numbers. To try things out without firing up the OSCBroker℠, skip to step 2 (section 3.2) below and install the OneStep Charge℠ source code.

It is recommended that you simply run the OSCBroker℠ JAR file from its default location($OSC/broker/OSCBroker.jar). However, you may certainly copy OSCBroker.jar to another location if you need to. If the operating system of the machine(s) upon which you are

running OneStep Charge℠ does not provide a sufficiently modern Java release (1.4.0 or greater), you may copy the OSCBroker.jar file to a host on your local network that does support a modern Java version. OneStep Charge℠ can easily be pointed toward a separate host via the OSCConfig.brokerHost parameter in $OSC/tri8osc/oscapi . By default, the live configuration is recommended to connect to the OSCBroker℠ on "localhost," but you can change this to be
any hostname or IP address (hostname is highly recommended) corresponding to the host upon which you have chosen to run the OSCBroker℠.

> **<u>IMPORTANT</u>**
> **PLEASE BE ADVISED THAT THE TCP/IP COMMUNICATIONS BETWEEN ONESTEP CHARGE℠ ITSELF AND THE OSCBROKER℠ <u>ARE NOT ENCRYPTED</u>**. Therefore, if you choose to run the OSCBroker℠ on a different host than the OneStep Charge℠ Progress code, such that OSCConfig.brokerHost is not "localhost" but is some other address, you MUST be certain that the network between OSC and the OSCBroker℠ is trusted, private, and free from eavesdropping/network sniffing or other attacks.

Run the broker by typing the following at a command prompt (DOS or Unix/Linux command prompt):

```
prompt> java -jar OSCBroker.jar &
```

Use the "&" (on Linux, for example) to send the OSCBroker℠ to the background and run it independently from the shell in which you executed the command. You may need to fully-qualify the "java" command or the location of the JAR file, depending upon your PATH environment variable.

For example:

```
prompt> /path/to/your/jre/bin/java -jar /path/to/OSCBroker.jar &
```

Also, please note that, depending on how you're logged in and what operating system you're running, exiting your shell might kill the Java process you just started. Be sure to check that. If it's a problem, you may want to use the “nohup” command (on most Unix/Linux OSes), for example, to prevent your telnet or SSH session from hanging up your Java process when you exit (if you're starting the broker while logged in via telnet or SSH, as opposed to a local console).

For example:

```
prompt> nohup java -jar OSCBroker.jar &
```

Set the host upon which you run OSCBroker℠ to run the OSCBroker℠ JAR file upon

system startup. This can be done with the command line placed in standard startup scripts specific to your Unix/Linux system, such as /etc/rc.local or in an init script of its own. On Windows, you may create a .bat file that runs the Java command (java -jar OSCBroker.jar), or use other such methods.

In all cases, make sure that your startup mechanism has appropriate PATH settings (or fully-qualify the "java" command) and access (permissions, etc.) to the OSCBroker℠ JAR file and the "java" command.

Please note that the OSCBroker℠ will, by default, output its standard log to OSCBroker.log in the current working directory in which you issue the "java" command. Be sure that proper permissions exist to write the log file. You may change the log file destination by using the “-l” command-line parameter.

To stop the broker, use the -stop command-line parameter (see below for more information on command-line parameters).

For example:

```
prompt> java -jar OSCBroker.jar -stop
```

It is very important to stop the OSCBroker℠ gracefully (using the -stop command instead of merely killing the Java process(es)), as doing so will allow the broker to finish all active transactions before shutting down.

The OSCBroker℠ will listen on TCP port 4888 by default (this can be changed with the -p command line parameter, see below). Be certain to allow this port, inbound, for any firewalls protecting the machine on which the OSCBroker℠ is running—allow inbound TCP port 4888 (or whatever port you choose to use instead via the -p command line parameter).

Please see the section in this guide concerning OSC Administrative Action Values for information about OSCBroker℠ diagnostics.

3.1.1 OSCBroker℠ Command-Line Parameters

The OSCBroker℠ supports some command-line parameters to aid in its execution on your host. General OSCBroker℠ usage is:

```
java -jar OSCBroker.jar [-l logFileName] [-p portNumber] [-v] [-stop]
```

Options:

- [-l logFileName]: If this parameter is specified, the OSCBroker℠ will write its log to the destination specified by logFileName instead of to the default “./OSCBroker.log”.
- [-p portNumber]: This parameter directs the OSCBroker℠ to attach to the port number

specified, instead of to the default (4888).

- [-v]: The OSCBroker℠ will simply display its version on standard output, then exit.
- [-stop]: The OSCBroker℠ will shut down. **Please note** that if you specified a different port number via the -p parameter when you initially started the broker, you must specify the port number again, via the -p parameter, when you issue the stop command. Example:
```
prompt> java -jar OSCBroker.jar -p 12345 -stop
```

3.1.2 OSCBroker℠ Single Settlement Module (OPTIONAL)

An additional, optional OSCBroker℠ jar file is also packaged with OneStep Charge℠: OSCBrokerSS.jar. This jar file can be used in place of the regular OSCBroker.jar file, using exactly the same methods, startup, and operation. You may rename the file to OSCBroker.jar if you wish to preserve existing startup scripts throughout your system, or you may choose to use OSCBrokerSS.jar without renaming it (e.g. use “java -jar OSCBrokerSS.jar” to start it instead of “java -jar OSCBroker.jar”). Be careful to keep the two jar files distinguishable, however, so that you can tell which one you are actually using.

Using OSCBrokerSS.jar instead of OSCBroker.jar will cause OneStep Charge℠ to operate in “single settlement” mode, whereby all transactions will be sent to **one** single processing node. **Triple-redundant failover and load balancing via the OSC Engine's standard network will not occur if OSCBrokerSS.jar is used.** This means that only one settlement batch per day will be generated by the OSC Engine, instead of up to N actual batches (one for each of N nodes, whichever nodes are actually utilized by the OSCBroker℠ during a given day).

For more information on settlement and why you may wish to use OSCBrokerSS.jar instead of OSCBroker.jar, see the “Transaction Settlement” section in this guide. It is recommended that OSC users use the standard OSCBroker.jar, and this is what most users prefer.

3.2 Installing the OneStep Charge℠ Progress Code

The OneStep Charge℠ Progress software is located in the "tri8osc" directory. In order to use OSC you will need to place the tri8osc directory somewhere in your PROPATH. This can be done by modifying your PROPATH to include your $OSC/tri8osc directory, or you may copy/move the tri8osc directory into a location already included by your existing PROPATH. This latter method eliminates the need to make changes in startup scenarios and parameters and is generally the easiest approach.

In addition, be sure to consult section in this guide concerning the OSC runtime configuration file. This file allows you to compile your default OSC configuration settings into your Progress R-Code executables but override some (or virtually all) settings at runtime on an as-needed basis, without the need for recompiling, such as when extra debug/diagnostic information is desired immediately.

If you can run only R-Code, even while testing OneStep Charge℠, and cannot also run compile-on-the-fly source code, then compile the OneStep Charge℠ core run program ($OSC/tri8osc/oscengin.p). This is the only file you need to compile. For live runtime environments, which are assumed to be R-Code only, be sure to compile $OSC/tri8osc/oscengin.p and make the resulting oscengin.r file available to your application.

COMPILING $OSC/tri8osc/oscengin.p into .r code, rather than letting Progress compile it for you on-the-fly each time you run it (during your evaluation of OSC) should cause OSC to run significantly faster.

4.0 Configuration

OneStep Charge℠ is configured via the OSCConfig table. The values contained in the OSCConfig temp-table are established in two basic ways: the values specified for OSCConfig in $OSC/tri8osc/oscapi, and the runtime overrides found in the configuration file.

4.1 OSCConfig Temp-Table

First, you may (and should) edit the assignments to the OSCConfig table in $OSC/tri8osc/oscapi and use values that suit your needs. These values will then be available in the OSCConfig table anywhere in a program after you include the API file (tri8osc/oscapi) in your code.

1. OSCConfig.confFile: Your configuration file location. Be sure that this file is readable (e.g. correct permissions) for any user or process which needs to use OneStep Charge℠. If the file is not readable, your runtime configuration overrides will not function properly nor reliably.

2. OSCConfig.serialNo and OSCConfig.licenseKey: OneStep Charge℠ serial number and license key.

3. OSCConfig.brokerHost and OSCConfig.brokerPort: Your OSC broker (OSCBroker℠) connection parameters. It is **highly** recommended that you use a host name (with accompanying DNS or hosts file) for your OSCConfig.brokerHost instead of an IP address.

4. OSCConfig.username and OSCConfig.password: Your OneStep Charge℠ username and password for running transactions. These values are assigned to you by Tri-8 when you license OneStep Charge℠ for live use. It is **highly** recommended that you do **not** specify your username and password in either your tri8osc/oscapi file nor your (perhaps world-readable) configuration file. Ideally, storing this data in your own database table and then assigning it to the OSCConfig table within tri8osc/oscapi is the best and most secure way to protect this sensitive information.

Those fields are the minimum fields required to get OSC running. Following are descriptions of the other fields which may be configured (see the file tri8osc/oscapi itself for inline comments).

5. OSCConfig.panicDir: Directory into which to place panic (save) files. This directory must be readable and writeable by any user who uses OneStep Charge[SM] if you wish to use the panic infrastructure. See the panicSave() function in the API Functions section for more information.

6. OSCConfig.defaultTimeout: This is the default timeout for basically all operations utilizing the OSC Engine. Please note that this is an approximation, in seconds, of the total execution, but actual run time can vary a bit. When creating a standard trExec record, you may specify the timeout for the respective operation by assigning trExec.timeout. This field of the OSCConfig table gives you an easy way to set defaults in your application programs—you can always just assign OSCConfig.defaultTimeout into your trExec.timeout field, if you want. **Note** that it is the trExec record which contains the effective timeout—the OSCConfig.defaultTimeout field merely gives you a place to save a default value which must still be assigned to individual trExec records. If you do not specify a value for the trExec.timeout field for any individual trExec record, OSCConfig.defaultTimeout will be used automatically for that record by OSC.

7. OSCConfig.yearOffset: This is used to offset years for two-digit-year expiration dates, and will likely never be changed.

8. OSCConfig.debug: This flag sets the logging verbosity of the OSC Engine. If set to true, OneStep Charge[SM] will be much more verbose in the information it logs and provides during program execution. It can be quite handy to toggle this flag in your config file to immediately activate debug verbosity on-demand without recompiling **any** programs, in order to solve a problem.

9. OSCConfig.vDebug: Reserved for use by Tri-8, Inc.

10. OSCConfig.devMode: The devMode field allows you to redirect OSC operations at runtime based upon an environment variable. If your Progress session has the environment variable "DEVMODE" set to "yes" then OSC will operate in development mode. The only difference is that OSC will read the devConfFile (see next item) instead of your standard config file, and it will use OSCConfig.devLogSpec and OSCConfig.devLogMethod for logging instead of OSCConfig.logSpec and OSCConfig.logMethod. Therefore, you can set a completely different brokerHost/brokerPort/username/password for development transactions via a separate config file, and merely have your startup environment set a variable in order to redirect transactions to a test merchant account. You can then easily test your application code without worrying about charging a live credit card.

11. OSCConfig.devConfFile: Exactly like OSCConfig.confFile, except that the file specified will be read instead of the one specified in OSCConfig.confFile if OSCConfig.devMode is toggled to true.

12. OSCConfig.logMethod: The logMethod field specifies the mechanism whereby OneStep Charge[SM] will deliver its log entries. The value for this field must be either "to" or "thru". If it is "to" then OSC will simply open the file specified in the logSpec field and deliver log lines to that file, corresponding basically to a standard Progress "output to" statement. If the value is "thru" then OSC will use the logSpec field to specify a command through which to deliver log lines, just like a standard Progress "output through" statement.

13. OSCConfig.logSpec: This field corresponds with the logMethod field (see description above). If logMethod is "to" then this field specifies the file to use for a log. For a "thru" logMethod this field specifies a program name/command. For example, with "to" as the logMethod this field might have a value of "/var/log/cc.log". For a logMethod of "thru" the logSpec might be "/usr/bin/logger -p local3.info" in order to use the UNIX Syslog daemon for logging OSC log lines via the local3.info facility.

14. OSCConfig.devLogMethod: This field is just like logMethod, except that it applies when OSCConfig.devMode is set to true. This allows you to use a separate logfile or logging facility when in development mode.

15. OSCConfig.devLogSpec: This field corresponds to devLogMethod in the same manner that logSpec corresponds to logMethod. Used only when OSCConfig.devMode is true.

4.2 Configuration File

Second, and in addition to the compiled-in defaults that you may specify in tri8osc/oscapi, you may edit the configuration file (the location of which is specified in OSCConfig.confFile). This file is read at runtime, so that values read therein override the compiled-in defaults that you specified in tri8osc/oscapi. A sample configuration file is provided in $OSC/conf/osc.conf. You may use this file as a guide for configuring runtime configuration overrides. The file includes inline comments for your assistance, but the basic format is var=value pairs, one pair per line, where "var" is the name of an OSCConfig field. For example, "debug=yes" would set the OSCConfig.debug field to true at runtime.

5.0 OneStep Charge[SM] API Specification

The OneStep Charge[SM] API is the interface through which you can harness the power of OneStep Charge[SM] and deliver it seamlessly to your application. The API itself requires very little effort to use with any Progress program. By including the standard API include file ($OSC/tri8osc/oscapi) in your program, you have all the infrastructure you need to use OneStep Charge[SM].

5.1 trExec Temp-Table

The OneStep Charge[SM] API is built around the trExec temp-table. This table is the transport mechanism whereby you can pass information to the OSC Engine and receive results back. The trExec table contains the following fields (data types in parentheses, maximum data length/value, where applicable, in brackets):

Field	Type	Limit
username	(char)	
password	(char)	
transNo	(int)	
enteredBy	(char)	**[maximum length: 20]**
trackData	(char)	
cardNo	(char)	**[maximum length: 16]**
expDate	(char)	**[maximum length: 4]**
amount	(dec)	**[maximum value: 999999.99]**
currencyCode	(int)	
CHName	(char)	**[maximum length: 60]**
CHAddress	(char)	**[maximum length: 80]**
CHAddress2	(char)	**[maximum length: 80]**
CHCity	(char)	**[maximum length: 40]**
CHState	(char)	**[maximum length: 20]**
CHZip	(char)	**[maximum length: 20]**
CHPhone	(char)	**[maximum length: 30]**
CHEmail	(char)	**[maximum length: 50]**
AVSResult	(int)	
CVV2	(char)	**[maximum length: 4]**
PONo	(char)	**[maximum length: 17]**
tax	(dec)	**[maximum value: 999999.99]**
shipToZip	(char)	**[maximum length: 10]**
vatNo	(char)	**[maximum length: 14]**
commodityCode	(int)	**[maximum value: 9999]**
discount	(dec)	**[maximum value: 9999999999.99]**
shipHandling	(dec)	**[maximum value: 9999999999.99]**
duty	(dec)	**[maximum value: 9999999999.99]**
numLines	(dec)	**[maximum value: 99]**
productCode	(char)	**[maximum length: 20] {EXTENT 99}**
price	(dec)	**[maximum value: 99999999.99] {EXTENT 99}**
lineDiscount	(dec)	**[maximum value: 9999999999.99] {EXTENT 99}**
quantity	(int)	**[maximum value: 999] {EXTENT 99}**
description	(char)	**[maximum length: 47] {EXTENT 99}**
uom	(char)	**[maximum length: 12] {EXTENT 99}**
routingNo	(char)	**[maximum length: 9]**
accountNo	(char)	**[maximum length: 17]**
OSCID	(char)	
batchNo	(int)	
authStatus	(int)	
authDetails	(char)	
apprCode	(char)	

authDate	(date)	
authTime	(int)	
settleDate	(date)	
settleTime	(int)	
comment	(char)	**[maximum length: 60]**
source	(char)	**[maximum length: 40]**
action	(int)	
timeout	(int)	
qFromDate	(date)	
qFromTime	(int)	
qToDate	(date)	
qToTime	(int)	
vaultID	(char)	
stored	(log)	
storageDate	(date)	
storageTime	(int)	
unStorageDate	(date)	
unStorageTime	(int)	
lastUpdateDate	(date)	
lastUpdateTime	(int)	
lastTransDate	(date)	
lasttransTime	(int)	
transCount	(int)	

Following are the descriptions of these fields.

1. trExec.username: Use the username field to specify a username specific to the individual trExec record, rather than accepting the username configured in OSCConfig.username. This is useful for running in multi-merchant environments. **You do not need to specify the trExec.username field in general**—use it only if you choose or if you are performing multi-merchant functions.

2. trExec.password: Like username, except that this specifies the password to use for this individual trExec record. Used for multi-merchant functionality.

3. trExec.transNo: The transNo field represents the primary tracking mechanism for payment transactions. It is user-assigned (programmer-assigned) and should always be a unique value within a single batch of transactions (trExec records) and, ideally, always unique within your entire system. It is used to retrieve status values and perform further processing, and therefore should **never be duplicated**.

4. trExec.enteredBy: This field allows you to track the author of a transaction through OSC. You may wish to assign this field the userid of the user who executed the original transaction, or something similar. This is useful for tracking later, especially if any problems arise which must be tracked to a specific user, and for audit purposes.

5. trExec.trackData; Full, unaltered track data (from a card swipe). If this field is not populated for new (CCA_AUTHONLY, CCA_PURCHASE, etc.) transactions then trExec.cardNo and trExec.expDate must be populated instead. This field is only used in card-present environments where a card swipe is done. OneStep Charge^SM will also parse and use individual tracks (1 and 2), so if you supply only track 1 or only track 2 data in the trExec.trackData field then the data will be sent appropriately.

6. trExec.cardNo: This is the credit/debit/private/etc. card account number to charge/credit. No dashes nor spaces allowed.

7. trExec.expDate: This is the expiration date of the credit card, in a "MMYY" format.

8. trExec.amount: The amount should be in dollars and cents (for US Dollar transactions) or in corresponding denominations for other currencies. An amount of "2" will be billed as "2.00" and is acceptable. Amounts "1.5" and "1.50" will both be processed as "1.50" .

9. trExec.currencyCode: The currency code for the transaction, which will interpret the amount value used. See the section in this guide concerning currency codes for values defined.

10. trExec.CHName: The CHName field specifies the cardholder name. If this field is non-blank, it will be transmitted as part of AVS (Address Verification Service) operations, which may yield better credit card processing fee rates, depending upon your merchant account setup.

11. trExec.CHAddress: This is the cardholder address. Like the CHName field, if this field is non-blank it will be transmitted as part of AVS operations.

12. trExec.CHAddress2: This is the cardholder address line 2. Like the CHAddress field, if this field is non-blank it will be transmitted as part of AVS operations.

13. trExec.CHCity: This is the cardholder city, (optionally) used for extra tracking/storage. **Please note that this field does not affect AVS**.

14. trExec.CHState: This is the cardholder state, (optionally) used for extra tracking/storage. **Please note that this field does not affect AVS**.

15. trExec.CHZip: This is the cardholder zip code, which functions like CHName and CHAddress, and **does** affect AVS. It will be transmitted for AVS if a value is supplied.

16. trExec.CHPhone: This is the cardholder phone number, (optionally) used for extra tracking/storage. **Please note that this field does not affect AVS**.

17. trExec.CHEmail: This is the cardholder email address, (optionally) used for extra tracking/storage. **Please note that this field does not affect AVS**.

18. trExec.AVSResult: The AVSResult field returns the specific level of address match achieved by the AVS attempt (if applicable). See the section in this guide concerning AVS result codes for values defined.

19. trExec.CVV2: The CV/CVV2 value is a special (usually three- or four-digit) value assigned to a credit card account to help counter fraud. It often appears on the back of the actual credit card (front for American Express cards). If this field is non-blank, it will be transmitted as part of the transaction (if applicable). The benefit to the merchant varies depending upon merchant account setup.

20. trExec.PONo: This field is used only when Purchase Card Level II (PC II) or Level III (PC III) support is desired. This field contains the purchase order number for the transaction, as you wish for it to appear. **NOTE** that only the first 17 alphanumeric characters are accepted by the processing networks, so if you supply more than 17 characters the value will be truncated. **PC II and PC III transactions only.**

21. trExec.tax: This field works like the trExec.amount field. The amount of sales tax for the transaction should be supplied via this field. If no tax is applicable, supply 0 (even on a PC II or PC III transaction). **PC II and PC III transactions only.**

22. trExec.shipToZip: The ZIP code to which the goods will be shipped, if any shipping is to occur. **PC II and PC III transactions only.**

23. trExec.vatNo: The VAT Registration number associated with the customer, if applicable. **PC III transactions only.**

24. trExec.commodityCode: The four-digit commodity code for the order, if any. **PC III transactions only.**

25. trExec.discount: The overall discount applicable to the order (the whole order), if any. **PC III transactions only.**

26. trExec.shipHandling: The shipping and handling total amount for the order, if any. **PC III transactions only.**

27. trExec.duty: The duty amount applicable to the order, if any. **PC III transactions only.**

28. trExec.numLines: The number of lines of order information that will be supplied on this Purchase Card Level III (PC III) transaction. Set this value to exactly the number of order lines for which you will supply PC III information. Note that if the number of order lines for which data is submitted exceeds the value of trExec.numLines, the extra lines will be ignored. **PC III transactions only.**

29. trExec.productCode[ ]: The item number (product code, SKU, etc.) of the particular order line. This field is an array (extent 99). Array element 1 represents the product code of the first order line (for this transaction). Array element 2 represents the product code for the second order line (for this transaction), and so on, up to a maximum of 99 order lines. If

any line data is supplied, productCode is a **required** field. That is, you cannot supply order line data and omit the productCode—either it must be supplied for the line or no line data must be supplied. **PC III transactions only.**

30. trExec.price[ ]: The price charged for this particular order line. Like trExec.productCode, this field is an array (extent 99) allowing the specification of up to 99 order lines. Array element N contains the price for order line N. If any line data is supplied, price is a **required** field. That is, you cannot supply order line data and omit the price—either it must be supplied for the line or no line data must be supplied. **PC III transactions only.**

31. trExec.lineDiscount[ ]: The discount amount applied for this particular order line. Like trExec.productCode, this field is an array (extent 99) allowing the specification of up to 99 order lines. Array element N contains the discount for order line N. **PC III transactions only.**

32. trExec.quantity[ ]: The quantity of goods for this particular order line. Like trExec.productCode, this field is an array (extent 99) allowing the specification of up to 99 order lines. If any line data is supplied, quantity is a **required** field. That is, you cannot supply order line data and omit the quantity—either it must be supplied for the line or no line data must be supplied. Array element N contains the quantity for order line N. **PC III transactions only.**

33. trExec.description[ ]: The line description for this particular order line. Like trExec.productCode, this field is an array (extent 99) allowing the specification of up to 99 order lines. Array element N contains the order line description for order line N. **PC III transactions only.**

34. trExec.uom[ ]: The unit of measure for this particular order line. Like trExec.productCode, this field is an array (extent 99) allowing the specification of up to 99 order lines. Array element N contains the unit of measure for order line N, such as "EACH" or "LBS" or "FEET." **PC III transactions only.**

35. trExec.routingNo: The routingNo field is used for ACH (e-check) transactions only. Supply the bank routing number here.

36. trExec.accountNo: The accountNo field is used for ACH (e-check) transactions only. Supply the bank account number here.

37. trExec.OSCID: This is the OneStep Charge℠ ID. It is a OneStep Charge℠-assigned value that uniquely identifies the transaction within OneStep Charge℠. Whereas the trExec.transNo field is user-assigned, the OSCID will be assigned by OSC and can be used for extended tracking or auditing. Please note that this field is quite important and should always be tracked (as should trExec.transNo, to be safe). The trExec.OSCID field is required to properly issue CCA_CAPTURE commands, as well.

38. trExec.batchNo: The trExec.batchNo contains settlement batch number. When a transaction has settled, CCA_STATUS requests on the transaction will populate the

trExec.batchNo field in their results, indicating the settlement batch of which the transaction was a part.

39. trExec.authStatus: The authStatus represents the current authorization status of the transaction. Authorization status values must be one of the pre-defined Credit Card Status values (beginning CCS_) or ACH Status values (beginning ACHS_), such as CCS_RTS for "Ready To Settle" or CCS_AUTHONLY for a card which has been authorized but not marked for inclusion in the open settlement batch.

40. trExec.authDetails: This field returns extended details about the transaction, as returned by either the processing network or OneStep Charge℠ itself (usually in the case of an error).

41. trExec.apprCode: This field indicates the approval code returned by the processing network, if applicable.

42. trExec.authDate: This field indicates the authorization request date, when applicable, **in Pacific Standard Time (Pacific Daylight Time during Daylight Saving Time periods) for all STATUS and QTRANS requests**. **Note** that this does not mean that the transactions was authorized (it might have been declined, for example), but rather this indicates the date on which the transaction was attempted. At transaction execution time, the value returns as, basically, the Progress "today" and "time" keywords, so the clock of the host running OSC will determine the value at execution time. Generally speaking, this is merely for backward compatibility—the authDate field is generally used later, for STATUS and QTRANS requests, when the OSC Engine timestamp (PST/PDT) will be used. Therefore, you may wish to apply proper timezone offsets for your OSC end users.

43. trExec.authTime: This field is like the authDate field, except that it represents the number of seconds since midnight (as in the 4GL "time" keyword) that the transaction was attempted. Using the authDate and authTime together, you can determine the particular second of time at which the transaction request was received. **PLEASE NOTE**, again, that at transaction execution time, "today" and "time" Progress 4GL keywords will be used to populate this field for backward compatibility, but the authTime field is generally used for STATUS and QTRANS operations and will be expressed in PST/PDT time in those cases.

44. trExec.settleDate: The settleDate indicates the date on which the transaction was settled (captured), in PST/PDT time.

45. trExec.settleTime: The settleTime corresponds with the settleDate and specifies the number of seconds since midnight that the transaction was settled. Using the settleDate and settleTime together, you can determine the particular second of time at which the settlement of the transaction occurred. Once again note that this value is PST/PDT time.

46. trExec.comment: The comment field allows you to track free-form user comments through OneStep Charge℠.

47. trExec.source: The trExec.source field provides another field into which your user-defined

tracking information can be inserted. This field can generally be populated with the source program name, or something similar, such as “ar/orderentry” or something else that you would like to track as the origin of the transaction request.

48. <u>trExec.action</u>: The action field specifies what type of transaction to perform against the information contained in the rest of the record. Some common actions are to authorize a card, return funds, check the status, and so forth. The action field must be assigned one of the pre-defined Credit Card Action (CCA) or ACH Action (ACHA) data members. These members are named beginning with “CCA_”, such as “CCA_AUTHONLY” for a transaction which preauthorizes a card (authorization only, without accompanying inclusion in the open settlement batch). A sample ACH action is ACHA_PURCHASE, to debit funds from a checking account (similar to a CCA_PURCHASE).

49. <u>trExec.timeout</u>: This field specifies the maximum allowable time, in seconds, to wait for the result of the action specified in the trExec.action field. Please note that this is merely an approximation that should fairly accurately be observed at runtime, under normal conditions. If you do not assign the trExec.timeout field, OneStep Charge[SM] will use the default timeout value found in OSCConfig.defaultTimeout.

50. <u>trExec.qFromDate</u>: This field allows you to filter a transaction query. The qFromDate represents the beginning date from which you wish to retrieve transactions.

51. <u>trExec.qFromTime</u>: This field allows you to filter a transaction query. The qFromTime represents the beginning time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) from which you wish to retrieve transactions.

52. <u>trExec.qToDate</u>: This field allows you to filter a transaction query. The qToDate represents the ending date to which you wish to retrieve transactions.

53. <u>trExec.qToTime</u>: This field allows you to filter a transaction query. The qToTime represents the ending time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) to which you wish to retrieve transactions.

54. trExec.vaultID: This field specifies the storage record in the OSC Vault[SM] upon which to operate. It can be used in place of sensitive cardholder data, such as cardNo and expDate, in order to retrieve said details from the OSC Vault[SM] for processing—it is a record key/identifier. **Requires OSC Vault[SM] subscription.**

55. trExec.stored: This field indicates whether the record represented is actively stored in the OSC Vault[SM] and available for use. **Requires OSC Vault[SM] subscription.**

56. trExec.storageDate: This field indicates the date upon which this record was stored in the OSC Vault[SM]. **Requires OSC Vault[SM] subscription.**

57. trExec.storageTime: This field indicates the time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) at which this record was stored in the OSC Vault[SM]. **Requires OSC Vault[SM] subscription.**

58. trExec.unStorageDate: This field indicates the date upon which this record was unstored from the OSC Vault℠. **Requires OSC Vault℠ subscription.**

59. trExec.unStorageTime: This field indicates the time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) at which this record was unstored from the OSC Vault℠. **Requires OSC Vault℠ subscription.**

60. trExec.lastUpdateDate: This field indicates the date upon which this record was last updated in the OSC Vault℠. **Requires OSC Vault℠ subscription.**

61. trExec.lastUpdateTime: This field indicates the time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) at which this record was last updated in the OSC Vault℠. **Requires OSC Vault℠ subscription.**

62. trExec.lastTransDate: This field indicates the date upon which this OSC Vault℠ record last had a payment transaction run against it. **Requires OSC Vault℠ subscription.**

63. trExec.lastTransTime: This field indicates the time (as an integer—the number of seconds since midnight, as the Progress “time” keyword operates) at which this OSC Vault℠ record last had a payment transaction run against it. **Requires OSC Vault℠ subscription.**

64. trExec.transCount: This field indicates the number of payment transactions that have been executed against the respective vaultID in the OSC Vault℠. **Requires OSC Vault℠ subscription.**

Note that many of these fields are completely optional. For example, only the following fields are required in order to process a credit card authorize-only or authorize-and-settle request:

transNo
trackData (or cardNo **and** expDate, or vaultID (requires OSC Vault℠ subscription)
amount
action

Providing other information can certainly be helpful and is often highly recommended.

In general, character fields whose values exceed the maximum listed length will be **truncated** before sending to the processor, but the transaction will still be sent. Numerical values, that is, integer and decimal fields, that exceed listed maximum values will result in an **error—**these fields are not modified by truncation or altered but, instead, the transaction is aborted with an error. The best practice is to ensure that your application trims values and respects the limits specified for data fields, as specified above.

5.2 Credit Card Action (CCA) Values

Each trExec record requires a value in the "action" field that determines the operation to execute on the rest of the data in the record. The set of possible action values for credit cards, defined in tri8osc/oscapi, begin with "CCA_" and are defined as integers. **Please note** that CCA actions are used for all "card types," including debit cards, private label cards, gift cards, and so forth. It is much more readable to use the named values via the API than to use the actual integer values. Following are the available actions and their descriptions. For each action, the minimum required fields for the trExec table are listed, as well as any additional supported fields. See the later section on Credit Card Status (CCS) values for descriptions of the CCS values. Where trExec.vaultID is indicated as a supported field, please note that an OSC Vault℠ subscription is required (vaultID applies only to subscribed OSC Vault℠ users).

1. **CCA_AUTHONLY**: Authorization only (pre-authorize).

   Required trExec fields: transNo
trackData (or cardNo **and** expDate, or vaultID)
amount
action

   Additional supported fields: timeout
currencyCode
CHName
CHAddress
CHAddress2
CHCity
CHState
CHZip
CHPhone
CHEmail
enteredBy
comment
source
PONo
tax
CVV2

   This action obtains a valid authorization for the credit card and decrements the cardholder's open-to-buy limit (available credit), but does not place the transaction in the open settlement batch for capture during the next scheduled settlement. This action is often used by merchants at order-entry time in non-retail environments, when shipping will occur at a later time. This transaction will hence secure the availability of funds, but further action must be taken in order to capture (settle) those funds so that they are actually transferred from the cardholder to the merchant's bank account. This further action is often taken at the time of shipment, in accordance association (Visa/MC/Amex/etc.) guidelines. A successful CCA_AUTHONLY transaction returns a trExec.authStatus value of CCS_AUTHONLY, indicating that the authorization succeeded but that the transaction is authorized only, as opposed to being ready to settle.

An AUTHONLY transaction must be captured (settled) manually within 14 days, or the authorization will expire, in which case you may re-run a new authorization.

2. **CCA_PURCHASE**: Authorize and prepare for settlement (SALE).

Required trExec fields:	transNo
	trackData (or cardNo **and** expDate, or vaultID)
	amount
	action
Additional supported fields:	timeout
	currencyCode
	CHName
	CHAddress
	CHAddress2
	CHCity
	CHState
	CHZip
	CHPhone
	CHEmail
	enteredBy
	comment
	source
	PONo
	tax
	CVV2

This action, like CCA_AUTHONLY, obtains a valid authorization for the credit card and decrements the cardholder's open-to-buy credit limit. However, unlike CCA_AUTHONLY, a CCA_PURCHASE transaction generally corresponds to a retail sale, where the entire transaction between a merchant and a customer is complete. The credit card transaction is included in the open settlement batch and will be settled during the next settlement execution, with funds thereafter being actually moved from the cardholder to the merchant. A successful CCA_PURCHASE transaction will return a CCS_RTS status value, indicating that the authorization succeeded and that the transaction is now ready to settle.

3. **CCA_CREDIT**: Credit/return funds.

METHOD 1: trackData or cardNo+expDate

Required trExec fields:	transNo
	trackData (or cardNo **and** expDate, or vaultID)
	amount
	action
Additional supported fields:	timeout
	currencyCode
	CHName
	CHAddress
	CHAddress2

CHCity
CHState
CHZip
CHPhone
CHEmail
enteredBy

This action reverses the normal flow of funds. Funds will be transferred from the merchant to the cardholder. This generally occurs during refund/return situations and to correct errors in billing. A successful CCA_CREDIT transaction returns a status of CCA_RTS, indicating that the transaction is ready to settle. Note that there is no such thing as an "authorization" for a credit. Unlike CCA_PURCHASEs and CCA_AUTONLYs, CCA_CREDIT transactions are placed immediately into the open settlement batch without any manner of contact with the cardholder's bank. The credit card processing industry does not support credit authorizations. Therefore, if there are problems with the card number (such as it being a closed account or bad number), then the problem will be seen at settlement time.

METHOD 2: OSCID

Required trExec fields:	OSCID (of a previous transaction) amount action
Additional supported fields:	transNo timeout

This alternative method of a CCA_CREDIT allows you to "refund" against a previously-authorized transaction, without having to know the credit card number and expiration date. This is a very common situation in many card processing environments. The OSCID specified should be the OSCID of a previously-authorized CCA_PURCHASE transaction or a CCA_CAPTURE transaction, and the specified amount will be credited back. You may run multiple credits (totaling up to the original captured amount).

In general, it is that simple. However, this operation does get a bit complicated inasmuch as it ties the CCA_CREDIT back to the original transaction, much like a CCA_CAPTURE. The METHOD 1 (cardNo+expDate) method of a CCA_CREDIT simply creates a brand new, standalone transaction. But this OSCID-based method links up all of the transaction operations in a "chain" which can get complicated if not very carefully managed—speaking of the CCA_QTRANS output which is designed to be "one row per transaction." The main points to remember are these:

(a) If your original transaction was a simple CCA_PURCHASE, you may simply CCA_CREDIT back an amount up to the original transaction amount, and subsequent CCA_QTRANS results will show the net amount. You may also issue multiple credits, up to the original amount.

(b) If your original transaction was a CCA_AUTHONLY, you MAY NOT credit against it unless it has been CCA_CAPTUREd. Once it has been CCA_CAPTUREd, you may credit, but **you must use the OSCID of the CCA_CAPTURE transaction in your CCA_CREDIT call.** This is very important. For example, if you CCA_AUTHORIZE a transaction and receive 023-123456 as your OSCID, then you CCA_CAPTURE against it and receive 023-456789 as the OSCID **of that CCA_CAPTURE** operation, then you must supply **023-456789** as the OSCID on your CCA_CREDIT call—you must credit **against the CCA_CAPTURE's OSCID, not against the CCA_AUTHONLY's OSCID.**

(c) Like a METHOD 1 CCA_CREDIT, this is an "offline" transaction. So, the OSC Engine will immediately accept it, provided all of the right fields are specified. However, **subsequently**, the OSC Engine may determine that the transaction is NOT valid—for example, the OSCID specified may be invalid (not pointing to any transaction in the system). The Engine will subsequently DECLINE the transaction and you would only discover this by CCA_STATUS/CCA_QTRANS. This is similar to CCA_CREDIT METHOD 1, which would only fail at settlement time if the cardNumber given turned out to be a closed/invalid account (for example), except that in the case of METHOD 2, the failure is known after mere seconds, instead of only at settlement time.

For more discussion of the nuances of the CCA_CREDITs using METHOD 2, particularly in the case of CCA_AUTHONLY with CCA_CAPTURE, please see the OSC Release Notes for OSC PCI 1.1.

4. **CCA_STATUS**: Check the status of a transaction.
   Required trExec fields: transNo
   action
   Additional supported fields: timeout

   This action allows you to check that status of a transaction you have previously executed. By supplying merely a transNo, action, and timeout, you may retrieve OneStep Charge[SM]'s view of the transaction, which is useful for ensuring that your application/database is up-to-date with the latest real-time status of transactions. Note that it is not particularly useful to check the status of transactions once they have reached a state where no more change is expected. For example, once a credit card transaction has been settled (and therefore moved to the CCS_CLOSED state), no more change is expected for the transaction—it is completed and closed. You may wish to use a CCA_STATUS operation in order to recall the settleDate or other information about the transaction, but the authStatus should not change. Note that only the last four digits of credit card numbers are returned.

   Transactions are available for querying or status checking (CCA_QTRANS or CCA_STATUS) almost immediately after they are executed. "Almost immediately," because there is a 5-10 second delay before the transactions are available in the querying infrastructure of the OSC Engine, once they execute.

5. **CCA_VOICEAUTH**: Issue (register) an offline authorization.
   Required trExec fields: transNo

trackData (or cardNo **and** expDate, or vaultID)
amount
apprCode
action

Additional supported fields:
timeout
currencyCode
CHName
CHAddress
CHAddress2
CHCity
CHState
CHZip
CHPhone
CHEmail
enteredBy
comment
source
PONo
tax
CVV2

The CCA_VOICEAUTH action allows you to register an offline authorization from your application. This is perhaps most commonly used in voice-authorization scenarios, when a CCA_PURCHASE (or CCA_AUTHONLY) transaction returns CCS_CALL, indicating that a voice-authorization must be obtained. The name CCA_VOICEAUTH may be slightly misleading, as you may attempt to "force" a transaction through without having literally obtained a voice-based authorization, but you will need a valid apprCode, and this is almost always observed in voice-based (manual) authorization scenarios. A successful CCA_VOICEAUTH transaction will return a status of CCS_RTS, indicating that the transaction is ready to settle.

6. **CCA_CAPTURE**: Mark a previously authorized-only (e.g. CCA_AUTHONLY) transaction as ready-to-settle.

Required trExec fields:
transNo
OSCID
action
amount

Additional supported fields:
timeout

This action corresponds to the additional step mentioned previously with regard to CCA_AUTHONLY transactions. After a card has been authorized via a CCA_AUTHONLY transaction, you must later capture (settle) the funds. This usually corresponds to the time of shipment for a product. In order to mark the previously-authorized transaction as ready-to-settle, you must supply the OSCID of the original transaction and an amount (plus the action, of course). This amount may be different than the amount that was authorized previously, which allows you flexibility in determining the final charge to the cardholder. However, be advised that problems may arise if you try to settle for a **greater** amount than

you authorized. That is not to say that no such transactions will succeed, but it is **highly** recommended that you instead estimate high at authorize time, and then settle for a possibly **lesser** amount at shipment time via your CCA_CAPTURE transaction.

The transNo field is "required" because it is used to index and retrieve trExec records, but it is the OSCID field that will specify which transaction to capture, so be sure that the OSCID is correct and refers to the exact previously-executed CCA_AUTHONLY transaction that you wish to capture.

Also note that authorizations are only good for a certain amount of time, which can actually fluctuate quite a bit depending on banks and even cardholders. Generally speaking, a credit card authorization is good for about 21 days. However, after 7-10 days that transaction is considered "stale" by most banks, and settling (capturing) the funds from that authorization may subject your merchant to an additional surcharge on the transaction. Therefore, it is highly recommended that you **minimize** the time between authorization and capture as much as possible, within your obvious merchant-specific constraints. OneStep Charge℠ imposes a 14-day authorization window. If an AUTHONLY transaction is older than 14 days, it cannot be captured via a CCA_CAPTURE command and is considered to be expired.

7. **CCA_STORE**: Store credit card information in the OneStep Charge Vault℠. **Requires OSC Vault subscription.**

Required trExec fields:	transNo
	cardNo **and** expDate (or vaultID)
	action
Additional supported fields:	timeout
	CHName
	CHAddress
	CHAddress2
	CHCity
	CHState
	CHZip
	CHPhone
	CHEmail
	comment
	source

This action stores sensitive payment card information in the PCI DSS-certified OneStep Charge Vault℠, for later retrieval. Using a CCA_STORE transaction, you may store information that you wish not to store in your own database, such as the credit card number and expiration date, cardholder name, etc., for later use, significantly reducing liability. The OneStep Charge Vault℠ will respond with a vaultID for the stored information, which you must record. Then, later, simply supply that vaultID on any Vault-supported payment transaction, without having to supply the respective data fields, and the OSC Vault℠ will retrieve those data fields using the vaultID, and insert them into the transaction as it runs through the OSC Engine℠. The entire process is seamless to your calling application—merely supply a vaultID and let the Vault℠ retrieve the sensitive, stored data

fields for cardNo, expDate, etc.

8. **CCA_UNSTORE**: “Unstore” credit card information in the OneStep Charge Vault^SM^.
Requires OSC Vault subscription.
Required trExec fields: transNo
vaultID
action
Additional supported fields: timeout

This action “unstores” stored payment card information from the OSC Vault^SM^. It is not exactly a full deletion, as certain information is retained for historical purposes. But, from the perspective of a calling application, unstoring information causes it to be retired.

9. **CCA_QTRANS**: Query OneStep Charge^SM^ Engine for transaction information.
Required trExec fields: action
Additional supported trExec fields: timeout
authStatus
cardNo
CHName
transNo
vaultID
qFromDate
qFromTime
qToDate
qToTime

The CCA_QTRANS action (along with CCA_QVAULT) is significantly different from other action values. Whereas other actions typically take a transaction (represented by a trExec record), perform the action on the transaction, and return the results via the same trExec record, the CCA_QTRANS action takes one trExec record (to use as the command record) and returns zero or more appropriate trExec records. This action is basically a reporting mechanism for OneStep Charge^SM^. The OSC user may use a CCA_QTRANS action to query transactions that are resident within the OSC infrastructure. For example, one might query OSC for the currently ready-to-settle transactions by issuing a CCA_QTRANS command, via a trExec record with the action set to CCA_QTRANS and the authStatus set to CCS_RTS. This tells OSC to return all transactions that are currently ready to settle. If qFromDate is supplied but qToDate is omitted (or vice versa), then qToDate (or qFromDate, respectively) will be set equal to qFromDate, so that the query will effectively target a single day's transactions. The trExec.qFromTime field defaults to 0 (midnight) upon creation (see $OSC/tri8osc/oscapi), and trExec.qToTime defaults to 86399 (23:59:59) **PST/PDT time.**

Please be aware that the indiscriminate use of the CCA_QTRANS command may result in **very large** record sets being returned, which may consume large amounts of network bandwidth and CPU time, depending upon your OSC architecture and configuration. For example, issuing a CCA_QTRANS command with **no** further criteria specified, such as

authStatus, will result in OneStep Charge℠ returning EVERY transaction in its database, regardless of status or date/time of execution. For the user who has been running OSC with a large number of transactions, this may be very wasteful and consume many precious resources. See the examples section for more information on how to use the CCA_QTRANS action appropriately.

CCA_QTRANS commands cannot span multiple merchant accounts, but must, rather, query one merchant account at a time (specified by username and password, as any trExec command would). As mentioned in the OSC introduction at the beginning of this User's Guide, **please also note** that historical transactions (CCS_DECL, CCS_CLOSED, CCS_ERR, CCS_CALL) **are not** guaranteed to be available via OneStep Charge℠ operations after a period of 120 days from settlement/declination.

10. CCA_QVAULT: Query OneStep Charge Vault℠ for stored payment card information.
Requires OSC Vault subscription.

Required trExec fields:	action
Additional supported trExec fields:	timeout
	cardNo
	CHName
	vaultID
	stored

The CCA_QVAULT action is similar to CCA_QTRANS inasmuch as it represents a query of the OneStep Charge℠ system, rather than an active transaction. OSC will return trExec records representing the results of the query. The CCA_QVAULT action will query the OSC Vault℠ for records stored there. See the section on OneStep Charge Vault℠ for more information about this valuable service. Using a CCA_QVAULT command, you may query the Vault℠ for records that are stored, or even “unstored” (inactive). You may filter by cardNo, CHName, and other such fields. For example, run a CCA_QVAULT with cardNo set to a credit card number to query the Vault℠ for all stored records for that credit card (in most cases there is no reason to store more than one single Vault℠ record for a particular credit card). Or, query with trExec.stored set to “yes” to return a list of all active Vault℠ records stored for the account. Query with trExec.stored set to “no” to return a list of all formerly stored records that are now “unstored” and inactive (no longer usable from the Vault℠).

5.3 ACH Action (ACHA) Values

The ACHA values, used for ACH transactions, are similar to their CCA counterparts, except that they route transactions through the ACH network and apply to bank accounts, as opposed to credit card accounts. Be advised, however, that ACH does not work the same way that credit cards do. Unlike the realtime authorization support that is available with credit cards, ACH is a much more asynchronous process. After an ACHA_PURCHASE or ACHA_CREDIT transaction executes (and returns ACHS_ACCEPTED), the transaction is no longer updated in the OSC Engine. The banks involved do not electronically communicate new information back to OneStep Charge℠ but, rather, will typically notify the merchant manually of any failures

“down the line” after submission, such as Insufficient Funds for a particular customer's account. The ACHA actions provide a means for ACH transaction submission via the Progress 4GL, but the normal level of “manual” ACH interaction that veteran ACH users are accustomed to should still be expected.

The available values are as follows.

1. **ACHA_PURCHASE**: Debit the checking account (like CCA_PURCHASE).

Required trExec fields:	transNo
	routingNo
	accountNo
	amount
	action
Additional supported fields:	timeout
	CHName
	CHAddress
	CHAddress2
	CHCity
	CHState
	CHZip
	CHPhone
	CHEmail
	enteredBy

   The ACHA_PURCHASE action will debit the bank account represented by routingNo and accountNo. A successful ACHA_PURCHASE transaction will return a trExec.authStatus value of ACHS_ACCEPTED to indicate that the transaction has been accepted into the ACH network. Otherwise, a status of ACHS_DECLINED will indicate the rejection of the transaction.

   There is no concept of AVS (Address Verification Service) with ACH, but the CHName, CHAddress, etc. parameters can be supplied nevertheless for tracking purposes.

   Note that the various “CH” (cardolder) parameters are supported even though there's no “card” involved. Think of the CHName as merely the general “account holder” name.

2. **ACHA_CREDIT**: Credit the checking account.

Required trExec fields:	transNo
	routingNo
	accountNo
	amount
	action
Additional supported fields:	timeout
	CHName
	CHAddress
	CHAddress2
	CHCity

CHState
CHZip
CHPhone
CHEmail
enteredBy

The ACHA_CREDIT action is just like ACHA_PURCHASE, except that funds move from the merchant to the customer.

3. **ACHA_STATUS**: Retrieve status of ACH transaction.
Required trExec fields: transNo
action
Additional supported trExec fields: timeout

This action allows you to check the status of a transaction you have previously executed. It is identical to CCA_STATUS, except that it applies to ACH transactions. Note that the account number (trExec.accountNo) is not returned. Note again that the status will not change even if an error such as Insufficient Funds occurs after ACH processing has begun —that information is not sent back to OSC by the financial institutions involved. The ACHA_STATUS command is for querying transactions about their submission results.

4. **ACHA_QTRANS**: Query OneStep Charge℠ Engine for ACH transaction information.
Required trExec fields: action
Additional supported trExec fields: timeout
authStatus
CHName
transNo
qFromDate
qFromTime
qToDate
qToTime
username
password

The ACHA_QTRANS action is identical to the CCA_QTRANS action, except that it applies to ACH transactions instead of credit/debit/etc. card transactions.

5.4 OSC Administrative Action (OSCA) Values

OneStep Charge℠ supports administrative actions to retrieve information about OSC itself or perform diagnostics functions. **Please note** that if you supply an OSCA trExec record in an oscengin.p call, **all other trExec records in the table for that call will be ignored**. That is, you cannot create two trExec records, assign OSCA_VERSION to one of them and CCA_AUTHONLY to the other (to run an actual transactions), and then run the trExec record via an oscengin.p call. OSC will execute the OSCA action only and will ignore the CCA action, in this example.

11. **OSCA_VERSION**: Show OneStep Charge[SM] version information.
Required trExec fields: action

    The OSCA_VERSION action will return the OneStep ChargeSM version in the trExec.authDetails field, with a trExec.authStatus of CCS_RTS.

12. **OSCA_BROKERCHK**: Test the OSCBroker[SM].
Required trExec fields: action

    This action will run basic diagnostic tests on the OSCBrokerSM to verify that it is functioning and set up correctly. The tests typically take about 10-15 seconds to complete. A value of CCS_RTS will be returned if the broker passes the the diagnostic tests. Otherwise, CCS_ERR will be returned. See trExec.authDetails for further information. In addition, OSC will use a separate log file during the OSCA_BROKERCHK process. The value ".brokerChk" will be appended to the current OSC log file name (logSpec or devLogSpec). For example, if you have specified "/var/log/cc.log" as your log file, then OSC will log to "/var/log/cc.log.brokerChk" during an OSCA_BROKERCHK operation. This applies only to static log files (OSCConfig.logMethod = "to"). If you are logging "through" a command, such as to pipe log messages through syslog, then no separate log will be generated (the OSCA_BROKERCHK logging information will appear in the same log file as all of your other OSC logging information).

5.5 Credit Card Status (CCS) Values

Like the Credit Card Action (CCA) values, the Credit Card Status (CCS) values are defined in the API interface file itself (tri8osc/oscapi). These values begin with "CCS_" and are defined as integers. Following is the list of possible CCS values and their descriptions.

1. **CCS_NODO**: No action taken (no status to report).

    This status value is more or less a marker status for new records in certain situations. It indicates that no action has been taken for the transaction.

2. **CCS_AUTHONLY**: Authorized only.

    This status indicates that the credit card has been authorized (only) but is not included in the open settlement batch.

3. **CCS_RTS**: Ready to settle.

    This status indicates that the transaction has been included in the open settlement batch and is ready to settle during the next batch settlement.

4. **CCS_DECL**: Card declined.

This status indicates that the transaction was declined by the processing network. The reason for the denial may be found in the trExec.authDetails field, though it is quite common for this field to yield little if any additional information. The reasons for declining credit card transactions vary greatly, from insufficient credit or open-to-buy limit to reported stolen cards.

5. **CCS_CLOSED**: Transaction settled.

   This status indicates that the transaction has been settled (captured) and is therefore closed to further processing.

6. **CCS_OK**: Operation succeeded.

   This status indicates that a particular non-payment transaction succeeded. It is typically used for CCA_STORE and CCA_UNSTORE actions.

7. **CCS_ERR**: A processing network error or OneStep Charge[SM] error has occured.

   This status indicates that the transaction was subject to an error. More specifically, either the processing network returned an error status or OSC itself did. The trExec.authDetails field is particularly useful for obtaining additional details in these cases.

8. **CCS_CALL**: Card requires voice authorization.

   This status indicates that the transaction requires manual voice authorization. Voice authorization is generally obtained via a phone call to the appropriate phone number, given to the merchant. Note that once this status is received the transaction is basically finished. You may establish a new transaction and issue a CCA_VOICEAUTH command on it, supplying your obtained approval code (apprCode), but the transaction that was marked as authStatus CCS_CALL is more or less finished at this point and will remain CCS_CALL. You may want to represent this as CCS_DECL (manually) in your database from this point forward.

9. **CCS_SYSERR**: System error occurred.

   This status indicates that the transaction subject to a system error. Unlike CCS_ERR, this does not mean that the processing network or OSC returned an error but rather that communication or Progress errors occurred. For example, if a network connection to the OSCBroker[SM] could not be established, CCS_SYSERR will be returned. Use the trExec.authDetails field for additional information. In general, CCS_SYSERR is returned when errors occurred **before** the transaction was delivered to the OSC processor for execution.

10. **CCS_SUBERR**: A severe system error occurred (error after submission).

    This status indicates that the transaction encountered an error, in a very similar fashion to CCS_SYSERR. However, unlike CCS_SYSERR, this error indicates that the transaction

appears to have been delivered to the OSC Engine, and therefore much care must be taken in solving the problem, as the card may have been charged. Think of this like a print spooler. Once data has been delivered to the printer, if an error occurs one must investigate carefully to determine if anything was actually printed on the printer. If it was, there is no way to "undo" this fact, though other corrective action may be taken. CCS_SUBERR is encountered, for example, when a timeout occurs in waiting for results from the OSC Engine. Be sure to set your trExec.timeout value high enough, though generally execution should not take more than several seconds.

5.6 ACH Status (ACHS) Values

The ACHS (ACH Status) values are very similar to CCS values, except that they apply to ACH transactions.

1. **ACHS_ACCEPTED**: The ACH transaction was accepted into the ACH network.

   This status value means that the transaction has not been rejected. It does not mean that the transaction **will** not be rejected. Since ACH is a very asynchronous process, especially as compared to credit card transactions, one must allow several days for funds to clear successfully. In general, though, ACHS_ACCEPTED means success for an ACH transaction.

2. **ACHS_DECLINED**: The ACH transaction has been declined or otherwise rejected.

   This status value indicates that the ACH network (or the OneStep Charge℠ Engine itself) has rejected the ACH transaction. Look in trExec.authDetails for additional information about the rejection.

3. **ACHS_ERR**: The ACH transaction was subject to an error.

   See the trExec.authDetails field for more info about the error.

4. **ACHS_SYSERR**, **ACHS_SUBERR**: Identical to CCS_SYSERR and CCS_SUBERR, respectively.

   Since SYSERR and SUBERR are OSC communication-related errors, the values ACHS_SYSERR and ACHS_SUBERR are provided merely to provide some continuity. They are the same as CCS_SYSERR and CCS_SUBERR.

5.7 Currency Code (OSC_CURR) Values

OneStep℠ Charge supports various currencies via the trExec.currencyCode field. Possible

values for this field are listed here. **Please note** that you must ensure that your merchant account (and its processor, such as Vital) is able to take the currency you are trying to use—you cannot merely assign a different trExec.currencyCode value and immediately start transacting in alternate currencies.

Value	Description	Value	Description
OSC_CURR_ARP	Argentine Pesos	OSC_CURR_ATS	Austrian Schillings
OSC_CURR_AUD	Australian Dollars	OSC_CURR_BBD	Barbadian Dollars
OSC_CURR_BEF	Belgian Francs	OSC_CURR_BGL	Bulgarian Lev
OSC_CURR_BMD	Bermudan Dollars	OSC_CURR_BRR	Brazilian Real
OSC_CURR_BSD	Bahamian Dollars	OSC_CURR_CAD	Canadian Dollars
OSC_CURR_CHF	Swiss Francs	OSC_CURR_CLP	Chilean Pesos
OSC_CURR_CNY	Chinese Yuan Renmimbi	OSC_CURR_CSK	Czech (Republic) Koruna
OSC_CURR_CYP	Cyprian Pounds	OSC_CURR_DEM	German Deutschemarks
OSC_CURR_DKK	Kroner (Denmark)	OSC_CURR_DZD	Algerian Dinars
OSC_CURR_EGP	Egyptian Pounds	OSC_CURR_ESP	Spanish Pesetas
OSC_CURR_EUR	Euros	OSC_CURR_FIM	Finnish Markka
OSC_CURR_FJD	Fijian Dollars	OSC_CURR_FRF	French Francs
OSC_CURR_GBP	UK Pounds	OSC_CURR_GRD	Greek Drachmas
OSC_CURR_HKD	Hong Kong Dollars	OSC_CURR_HUF	Hungarian Forint
OSC_CURR_IDR	Indonesian Rupiah	OSC_CURR_IEP	Irish Punt
OSC_CURR_ILS	Israeli New Shekels	OSC_CURR_INR	Indian Rupees
OSC_CURR_ISK	Icelandish Rupiah	OSC_CURR_ITL	Italian Lira
OSC_CURR_JMD	Jamaican Dollars	OSC_CURR_JPY	Japanese Yen
OSC_CURR_KRW	South Korean Won	OSC_CURR_NLG	Dutch Gilders
OSC_CURR_RUR	Russian Rubles	OSC_CURR_SAR	Saudi Arabian Riyal
OSC_CURR_SDD	Sudan Dinar	OSC_CURR_SEK	Swedish Krona
OSC_CURR_SGD	Singaporean Dollars	OSC_CURR_SKK	Slovakian Koruna
OSC_CURR_THB	Thai Baht	OSC_CURR_TRL	Turkish Lira
OSC_CURR_TTD	Trinidad and Tobago Dollars	OSC_CURR_TWD	Taiwanese Dollars
OSC_CURR_USD	US Dollars	OSC_CURR_VEB	Venezuelan Bolivar
OSC_CURR_XAG	Silver Ounces	OSC_CURR_XAU	Gold Ounces
OSC_CURR_XCD	Eastern Caribbean Dollars	OSC_CURR_XDR	Special Drawing Right (IMF)
OSC_CURR_XPD	Palladium Ounces	OSC_CURR_XPT	Platinum Ounces
OSC_CURR_ZAR	South African Rand	OSC_CURR_ZMK	Zambian Kwacha

5.8 AVS Result (AVS) Values

The results of AVS operations indicate how precise the match was for AVS data submitted on a transaction. The AVS result code can be found in trExec.AVSResult after a relevant transaction returns.

AVS Code	Description
AVS_EXACT_ZIP9	Exact match, 9 digit zip code
AVS_EXACT_ZIP5	Exact match, 5 digit zip code
AVS_STREET_ONLY	Street address match only
AVS_ZIP9_ONLY	9 digit zip code match only
AVS_ZIP5_ONLY	5 digit zip code match only
AVS_NO_MATCH	No match on street address or zip code
AVS_UNAVAIL	AVS unavailable on this card
AVS_NONUS_UNAVAIL	Non-US card issuer, AVS unavailable
AVS_ISSUER_DOWN	Card issuer system (cardholder's bank) currently down, try again later.
AVS_ERROR_INELIGIBLE	Error, ineligible—not a mail/phone order
AVS_NOT_SUPPORTED	Service not supported
AVS_GEN_DECL	General decline or other error

5.9 API Functions

The following functions are defined within the OSC API to help with a few tasks.

1. function panicSave returns log (input table for trExec, input transNo as int)

   This function saves a trExec record to disk, at OSCConfig.panicDir/filename where filename is the transNo of the trExec record. The transNo is supplied as the second parameter. If transNo = -1, then the first record in the trExec table (and presumably the only, if transNo = -1) is written. This function is useful if you are encountering errors or want to make sure that, in certain situations, a Progress UNDO does not destroy information retrieved from the trExec table. This function is merely a means by which you can quickly export some/all records to disk temporarily.

2. function OSCConstChar returns char (input OSCConstant as int)

   This function returns a character representation of the OSCConstant (a CCA_/ACHA_/OSCA_ or CCS_/ACHS_ value), which is useful for display/logging purposes. For example, OSCConstChar(CCS_AUTHONLY) returns the character string "CCS_AUTHONLY" which is useful because the literal value of CCS_AUTHONLY is merely an integer.

3. function OSCConstEnglish returns char (input OSCConstant as int)

   This function returns an English description of the OSCConstant (a CCA_/ACHA_/OSCA_ or CCS_/ACHS_ value), which is useful for end-user displays. For example, OSCConstEnglish(CCS_AUTHONLY) returns "AUTHORIZED (ONLY)".

4. function OSCConstEnglishAbbr returns char (input OSCConstant as int)

   This function is just like OSCConstEnglish except that it returns fixed-width descriptions, which are useful when you have a fixed-width browse column, for example. All returned English descriptions are 8 characters or fewer. For example, OSCConstEnglishAbbr(CCS_AUTHONLY) returns "AUTHONLY".

5. function maskVal returns char (input inString as char, input numClear as int)

   This function masks all but the last numClear characters of inString with "X" characters. For example, maskVal("1234567890123456",4) yields "XXXXXXXXXXXX3456" .

The following three functions perform basically the same function, but can be used to standardize on masking policies by transaction type (which may change in future releases).

6. function CCMask returns char (input inString as char)

   This function uses the default usage of maskVal to mask a credit card number. For example, CCMask("1234567890123456") returns "XXXXXXXXXXXX3456" (all but last four digits of a credit card number are masked).

7. function ACHRoutingMask returns char (input inString as char)

   This function uses the default usage of maskVal to mask an ACH routing number. For example, ACHRoutingMask("12345678") returns "XXXX5678" (all but last four digits of a routing number are masked). Note that the OSC Engine itself does not mask routing numbers, but will return them in their entirety.

8. function ACHAccountMask returns char (input inString as char)

   This function uses the default usage of maskVal to mask an ACH account number. For example, ACHAccountMask("123456") returns "XX3456" (all but last four digits of a credit card number are masked).

5.10 Multi-Merchant Support

As of OneStep Charge[SM] 1.1, multi-merchant operations are supported directly in the trExec temp-table. By supplying the trExec.username and trExec.password fields on any trExec record, you will override the default OSC username and password for that particular transaction. This allows you to specify transaction-level multi-merchant information in a single

OneStep Charge[SM] engine call at runtime. You may, as always, build a trExec temp-table with many rows, but with the trExec.username and trExec.password fields you may book each transaction (each trExec record) in your table to a different merchant account, if you so desire. Note that the correct username and password must always be used to properly query transactions via CCA_STATUS and CCA_QTRANS.

If you do not supply a value for trExec.username and trExec.password, the defaults from OSCConfig.username and OSCConfig.password will be used. Therefore, trExec.username and trExec.password need only be used by those merchants who wish to utilize multi-merchant support in the trExec temp-table.

5.11 Purchase Card Level II (PC II) Support

Purchase Card Level II data provides many advantages. Two primary advantages are "trackability" and better interchange fees. A "Purchase Card" or "Corporate Card" is a special credit card. It operates in the same basic way as a credit card, but it is typically issued to corporate entities for use with company purchases. There is typically a daily spending limit, and that limit can be quite low. The US Federal Government, for one example, uses Purchase Cards (and other very similar formats) quite extensively. Since OneStep Charge[SM] supports PC II, you may provide three key pieces of extra data when you transact on a Purchase Card and reap the benefits of PC II. These three extra pieces of data are the purchase order number ("PO number") relevant to the transaction, the tax, and the ship-to ZIP code. Supply these values in trExec.PONo, trExec.tax, and trExec.shipToZip respectively. If the card is indeed a Purchase Card, and provided that your merchant account runs through a PC-II-supported processing network, you will receive PC II qualified interchange fees on the transaction (which are usually much lower than non-qualified fees), and the cardholder will receive extra transaction detail (including the PO number) on his end-of-month statement. Note that if trExec.tax and/or trExec.shipToZip do not apply (such as for digital products or non-taxed sales) they can be left blank, as the key field for PC II triggering is trExec.PONo. However, the trExec.shipToZip field is required for AMEX Purchase Corporate transactions, so be sure to supply it if possible, at least for AMEX PC transactions.

For an example PC II transaction, see the "Examples" section.

5.12 Purchase Card Level III (PC III) Support

Purchase Card Level III extends PC II by adding even more detail about the transaction, typically for use on the cardholder's statement to provide more detail for tracking expenses. OneStep Charge[SM] supports PC III for non-T&E, non-fleet transactions on the TSYS (formerly known as Vital) platform for CCA_PURCHASE transactions. To use PC III, supply values for any of the following fields:

trExec.vatNo
trExec.commodityCode
trExec.discount

trExec.shipHandling
trExec.duty
trExec.numLines

as well as any of the line-specific PC III fields:

trExec.productCode[ ]
trExec.price[ ]
trExec.lineDiscount[ ]
trExec.quantity[ ]
trExec.description[ ]
trExec.uom[ ]

Additionally, note that PC III transactions augment PC II transactions. That is, a PC III transaction can contain all of the PC II fields (such as trExec.PONo), as well as the additional PC III fields listed above.

For example PC III transactions, see the “Examples” section.

5.13 OneStep Charge Vault

The OneStep Charge Vault℠ is a value-added service available through the standard OSC API, which enables the storage of sensitive cardholder data without the accompanying liability and expense of Payment Card Industry Data Security Standard (PCI DSS) compliance. When a merchant or processing entity stores sensitive card data, such as the credit card number, expiration date, cardholder name and address, and so on, significant liability is incurred—and data breaches make headlines almost daily. With the OneStep Charge Vault℠, organizations can utilize OSC's DSS-certified secure storage to store data for later use. Many scenarios make storing credit card data necessary. For example, recurring, automated billing would require a merchant to keep cardholder data on file, yet that merchant may wish to avoid the liability and expensive of securing the sensitive fields. Similarly, a manufacturer who ships part of an order, and needs to capture a previous authorization (a CCS_AUTHONLY transaction created by a previous CCA_AUTHONLY command), yet also needs to re-authorize the remaining balance of the order back onto the credit card, would have a similar need to keep the credit card number on file, yet may want to avoid the liability of storing in on-site.

Using OSC Vault℠ functionality requires a distinct subscription, with the features activated on your account. Please contact us (see the “Getting Help” section) if you wish to activate OSC Vault℠ functionality on your account.

Basic Vault℠ Architecture

The OneStep Charge Vault℠ operates using a very simple store-use-unstore architecture. The essential process flow is:

1. Store the desired fields with a CCA_STORE command.

2. Record the vaultID returned by the CCA_STORE transaction into your own database, rather that storing the cardNo and expDate.
3. When a subsequent payment transaction is desired, supply the vaultID for the transaction (such as a CCA_AUTHONLY) instead of cardNo and expDate. The OSC Vault℠ will retrieve the cardholder data for the given vaultID and will automatically use that data for the payment transaction.
4. When the cardholder data is no longer needed, “unstore” it with a CCA_UNSTORE command, supplying the vaultID you wish to unstore. Unstoring data removes it from the Vault℠ and marks it as “inactive,” though a record of the original vaultID is retained.

Supported Fields for Vault℠ Storage

The following trExec fields may be stored in the OneStep Charge Vault℠:

cardNo
expDate
CHName
CHAddress
CHAddress2
CHCity
CHState
CHZip
CHPhone
CHEmail
comment
source

Utilizing and Overriding Vault℠ Fields

As the above documentation for CCA_AUTHONLY, CCA_PURCHASE, and other CCA transactions shows, the vaultID field may be supplied in place of cardNo and expDate. If it is, the Vault℠ will retrieve all stored fields for the vaultID and use them on the transaction. Any fields supplied on the transaction itself will **override** the Vault℠ fields. For example, if a given Vault℠ record has cardNo, expDate, CHName, CHAddress and CHZip stored, and a CCA_PURCHASE transaction specifies none of these fields (but specifies the vaultID), then the Vault℠ will supply all five fields. But if the CCA_PURCHASE transaction **does** supply CHName and CHZip, the values supplied will **override** the Vault-stored fields, for those fields only, for that particular transaction. That is, supplying a vaultID on a payment transaction will cause the Vault℠ **first** to fill in **all** of its stored fields for the vaultID, and then the supplied trExec fields will override any retrieved values for the given fields.

Updating Stored Values

To update the data stored in the Vault℠, run a CCA_STORE transaction, supplying the vaultID of the record to update, and any trExec fields that you wish to update. For example, if a cardholder has updated his expiration date, run a CCA_STORE, supply the vaultID of the Vault℠ record to update, and supply the new expDate (but leave all other trExec fields blank).

The OSC Vault℠ will update the stored data for expDate, leaving all other fields unchanged.

Removing a Record from the Vault℠ (Unstoring)

Use the CCA_UNSTORE action to "unstore" Vault℠ data. Running a CCA_UNSTORE, with a vaultID supplied, will cause the Vault℠ to retire the Vault℠ record and its stored data. The record itself will be retained for historical purposes, but the trExec.stored field will be set to "no" and the Vault℠ record can no longer be used on any payment transactions.

Querying the Vault℠

Use the CCA_QVAULT action to query the OSC Vault℠ for record information in the same essential manner that CCA_QTRANS can be used to query the OSC Engine for transaction information. This allows you to get a listing of your stored Vault℠ records. Because records are retained permanently even after an unstore (though cardholder data is no longer available for use and the Vault℠ record cannot be used on a payment transaction), you may use the trExec.stored field to query for stored, unstored, or all, records. For example, querying with trExec.stored set to "yes" will return all available, "live" (active) Vault℠ records, whereas leaving trExec.stored at its default "?" value will return ALL Vault℠ records, including unstored (inactive) records.

Note that trExec records returned by a CCA_QVAULT contain several handy fields that apply only to Vault records:

vaultID
stored
storageDate
storageTime
unStorageDate
unStorageTime
lastUpdateDate
lastUpdateTime
lastTransDate
lastTransTime
transCount

See the definitions of these fields (above) for further descriptions of their use.

See the "Examples" section for OSC Vault℠ example transactions.

5.14 Transaction Settlement

In the electronic payment processing world, no money actually moves until a "settlement" occurs, with almost all payment types. During a normal business day, a merchant builds an "open batch" by performing CCA_PURCHASE transactions, CCA_CREDIT transactions, CCA_CAPTUREs, and so forth. At the end of the day, the open batch must be presented to the processor to be delivered into the clearing infrastructure of the associations (Visa/MC,

Amex, etc.). This process is called “settling” the open batch. If a merchant fails to settle transactions promptly enough, surcharges are levied against the merchant. It is recommended to **always** settle at least once per day.

OneStep Charge℠ handles settlement automatically. The OSC Engine will settle all open transactions (all open batches—all CCS_RTS transactions) each day at 20:00 Pacific Time (8:00 p.m.). The settlement process also includes sending ACH transactions to the ODFI (Originating Deposit Financial Institution) and into the ACH network. After this time, new transactions will be included in the next day's batch.

After settlement occurs, which typically takes only a few seconds once it has begun, your settled card transactions will have a status of “CCS_CLOSED” and will have a trExec.settleDate, trExec.settleTime and trExec.batchNo when you query them (via CCA/ACHA_STATUS or CCA/ACHA_QTRANS). See the section in this guide concerning ACH for information on determining the success of ACH transactions. Any transactions that fail during the settlement process will NOT receive a batchNo. OneStep Charge℠ automatically works with the processing network to solve batch settlement issues and get them resolved. Please note that settlement issues/problems are very rare, largely thanks to the authorization infrastructure present in the industry and OSC's use of it.

It is important to note that in the electronic payments industry, actual bank deposits are asynchronous and are subject to various delays (banking holidays, card association rules, Federal Reserve Bank schedules, etc.). If a bank settlement occurs via OneStep Charge℠ on Monday, August 1, the funds are typically in process for 2-4 days, and the merchant will typically receive the **actual deposit** on, say, Thursday, August 4 or Friday, August 5. Some merchants have agreements that guarantee that their settlements are funded faster—this is entirely up to the merchant's bank and has nothing to do with OSC itself. Once OSC has submitted the settlement batch, it is subject to the same delays throughout the banking system that any other credit card settlement batch (regardless of software used) is subject to.

To further complicate the issue, it is common for a merchant's merchant account to have separate American Express (AMEX) deposits vs. Visa/Mastercard. So, for example, a settlement batch that has ten $10 transactions in it (4 AMEX and 6 Visa/MC) may produce **two** deposits: AMEX will deposit their funds according to their schedule or agreement, and Visa/MC (that is, the acquiring bank) will deposit theirs, independent of the others. If this example transaction from Monday, August 1, had the aforementioned ten $10 transactions, then the $40 worth of AMEX may arrive in the merchant's checking account on Wednesday, August 3 while the remaining $60 arrives Thursday or Friday. All of this, again, is completely independent of OneStep Charge℠.

One factor of OneStep Charge℠ itself, however, must be considered, and that is the triple-redundant failover infrastructure that OSC employs at the OSC Engine. Currently there are three processing nodes (see Figure 1), but this number may increase or decrease, with no action necessary by the OSC user. Each node is entirely independent of the other nodes, and can process entirely by itself. Thus, if one node fails or is shut down, or the merchant's network route to that node is interrupted, processing continues uninterrupted using the other remaining nodes. This also provides automatic load balancing. A side effect of this

architecture is that each processing node submits its own settlement batch at the end of each day. Thus, if, during a day's worth of processing, OSC transactions sent through the OSCBroker℠ hit **three** different nodes (which occurs according to availability and load, as well as random factors), then on that day **three** actual settlements will be submitted. In the ten-transaction example, if two transactions hit Node 1, three hit Node 2, and five hit Node 3, then three actual settlements would occur, of two, three and five transactions respectively. While all of this complexity happens “behind the scenes,” completely unseen by the OSC user, it does mean that multiple bank deposits will occur (on top of the already-multiplied deposit schedule common to all credit card processing, described above). The trExec.batchNo fields for a given day's worth of transactions would show three different batch numbers, revealing that three actual settlements occurred (one from each node settling its transactions for the day).

For most OneStep Charge℠ users, the complexity of the triple-redundant infrastructure is happily hidden and automatic. However, some OSC users prefer to eliminate the multiple settlements that occur, usually in order to help simplify a bank deposit reconciliation process that is already quite complicated by standard credit card transaction settlement practices (independent of OSC, as described above). For this reason, the OSCBrokerSS.jar file is provided (OSCBroker℠ Single Settlement module). By using OSCBrokerSS.jar instead of OSCBroker.jar, all transactions will be forced to use **one** single processing node, and thus only one daily settlement will occur on the backend. **IMPORTANT: this means that if the one processing node is unavailable for any reason, transactions will fail until it becomes available again. There is no triple-redundancy when using OSCBRokerSS.jar.** It is worth noting that uptimes for each processing node are very high, so OSC users choosing to use OSCBrokerSS.jar typically experience almost no downtime, even without the redundancy, and find it to be an acceptable way to eliminate the extra complexity of multiple daily settlements.

5.15 Data Availability Delay

It is important to note that the OneStep Charge℠ Engine utilizes a complex distributed database to process transactions and then archive and index them for separate, long-term query availability. Generally speaking, when a transaction is run for the first time, it takes 5-10 seconds for the transaction to propagate to the OSC Engine archiving area for querying (ACHA/CCA_STATUS, ACHA/CCA_QTRANS). If you run a transaction and then immediately run, for example, a CCA_STATUS on the transaction, you may receive a “transaction not found” error. This only occurs when no pause of several seconds is inserted between the transaction run itself and the CCA_STATUS operation (in this example).

6.0 Examples

This section includes examples of OneStep Charge℠ API usage and execution. These simple examples should serve to clarify the API and give you a basic understanding of how to use OneStep Charge℠. Your user interface is up to you, of course, and these examples serve merely to illustrate the OSC backend functionality.

6.1 CCA_AUTHONLY Example

Here we'll see how to execute a basic authorization-only transaction.

```
/* include the OSC API in order to use OneStep ChargeSM */
{tri8osc/oscapi}

create  trExec.
assign  trExec.transNo = next-value(yourProgressSequence)
        trExec.cardNo  = "1234567812345678"
        trExec.expDate = "0305"
        trExec.amount  = 17.25
        trExec.action  = CCA_AUTHONLY
        trExec.timeout = 90.

run tri8osc/oscengin.p (input-output table trExec).
```

After the run statement returns, you may examine the trExec table to see the results of the transaction. Of particular interest is the authStatus field, which specifies whether or not authorization was obtained (in this example case of a CCA_AUTHONLY action). So, to expand upon the above snippet of code, one might operate OneStep Charge[SM] in the following simple manner.

```
/* see above code for trExec creation */
run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists, else you'd need "find first" */

if not available trExec
then do:
   /* error message, return, quit, etc. */
end.

case trExec.authStatus:
when CCS_AUTHONLY
then do:
   message "Card approved: " trExec.authDetails.
   /* register order, print work order, etc. */
end. /* when CCS_AUTHONLY */

when CCS_DECL
then do:
   message "Card declined: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* when CCS_DECL */
```

```
when CCS_ERR
then do:
   /* error message, return, etc. */
end. /* when CCS_ERR */
end case. /* trExec.authStatus */
```

For testing purposes, a test credit card number and specific expiration date and address values can be used. See the section in this guide concerning “Test Credit Card Numbers” for details.

6.2 CCA_PURCHASE Example

Using the CCA_PURCHASE action is virtually identical to using CCA_AUTHONLY, except that a different authStatus is expected.

```
/* include the OSC API in order to use OneStep ChargeSM */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo   = next-value(yourProgressSequence)
       trExec.cardNo    = "1234567812345678"
       trExec.expDate   = "0305"
       trExec.amount    = 17.25
       trExec.CHName    = “Joe Sakic”
       trExec.CHAddress = “123 Stanley Cup Place”
       trExec.CHZip     = “11111”
       trExec.enteredBy = “dan”
       trExec.source    = “oscexample.p”
       trExec.action    = CCA_PURCHASE
       trExec.timeout   = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
   /* error message, return, quit, etc. */
end.

case trExec.authStatus:
when CCS_RTS   /* <-- NOTICE this is not CCS_AUTHONLY */
then do:
   message “Card approved: “ trExec.authDetails.
   /* register order, print receipt, etc. */
end. /* when CCS_AUTHONLY */
```

```
when CCS_DECL
then do:
   message "Card declined: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* when CCS_DECL */
```

Notice that this example also supplied some additional fields to the trExec record. Namely, CHName, CHAddress and CHZip were supplied for AVS (Address Verification Service) operations, and the enteredBy and source fields were used for some extra tracking. These optional fields expand the usefulness of the OSC reporting mechanisms, as well as giving the merchant better auditing capabilities and possibly even significantly better credit card processing fees, thanks to AVS.

6.2.1 CCA_PURCHASE Example - Purchase Card Level II (PC II)

Here we will run a very similar CCA_PURCHASE, except that we will supply some extra fields that will cause a Purchase Card Level II (PC II) transaction to occur, which can yield better discount rates on credit card processing fees as well as providing more information to the cardholder on his/her statement.

```
/* include the OSC API */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo   = next-value(yourProgressSequence)
       trExec.cardNo    = "1234567812345678"
       trExec.expDate   = "0305"
       trExec.amount    = 17.25
       trExec.CHName    = "Joe Sakic"
       trExec.CHAddress = "123 Stanley Cup Place"
       trExec.CHZip     = "11111"
       trExec.enteredBy = "dan"
       trExec.source    = "oscexample.p"
       trExec.PONo      = "44567"
       trExec.tax       = 1.23
       trExec.shipToZip = "11112"
       trExec.action    = CCA_PURCHASE
       trExec.timeout   = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
```

```
then do:
   /* error message, return, quit, etc. */
end.

case trExec.authStatus:
when CCS_RTS   /* <-- NOTICE this is not CCS_AUTHONLY */
then do:
   message "Card approved: " trExec.authDetails.
   /* register order, print receipt, etc. */
end. /* when CCS_AUTHONLY */

when CCS_DECL
then do:
   message "Card declined: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* when CCS_DECL */
```

This example supplied the PONo (purchase order number), tax, and shipToZip fields for trExec. These fields are PC II fields and will be sent along with the transaction, typically qualifying it as a PC II transaction.

6.2.2 CCA_PURCHASE Example - Purchase Card Level III (PC III), No Lines

Here we will augment the CCA_PURCHASE PC II example above, adding even more fields, to run a Purchase Card Level III transaction. PC III supplies even more information to the cardholder's bank for inclusion on a statement, and is used for reconciling expenses with more accuracy and clarity.

```
/* include the OSC API */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo       = next-value(yourProgressSequence)
       trExec.cardNo        = "1234567812345678"
       trExec.expDate       = "0305"
       trExec.amount        = 17.25
       trExec.CHName        = "Joe Sakic"
       trExec.CHAddress     = "123 Stanley Cup Place"
       trExec.CHZip         = "11111"
       trExec.enteredBy     = "dan"
       trExec.source        = "oscexample.p"
       trExec.PONo          = "44567"
       trExec.tax           = 1.23
       trExec.shipToZip     = "11112"
       trExec.vatNo         = "GB12345678"
       trExec.commodityCode = 1234
```

```
        trExec.discount      = 1.01
        trExec.shipHandling  = 2.12
        trExec.duty          = 1.04
        trExec.action        = CCA_PURCHASE
        trExec.timeout       = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
   /* error message, return, quit, etc. */
end.

case trExec.authStatus:
when CCS_RTS   /* <-- NOTICE this is not CCS_AUTHONLY */
then do:
   message "Card approved: " trExec.authDetails.
   /* register order, print receipt, etc. */
end. /* when CCS_AUTHONLY */

when CCS_DECL
then do:
   message "Card declined: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* when CCS_DECL */
```

This example supplied the PONo (purchase order number), tax, and shipToZip fields, which are PC II fields, but then also supplied vatNo, commodityCode, discount, shipHandling, and duty, all of which are PC III fields. Not every PC III must be supplied every time. The PC III fields supplied will be sent with the transaction, to qualify it as PC III and provide the additional data to the cardholder.

6.2.2 CCA_PURCHASE Example - Purchase Card Level III (PC III), With Lines

Here we will augment the CCA_PURCHASE PC III example above, adding even more fields for line-by-line order information data (order lines). You may supply order line information for up to 99 order lines on a single OSC transaction.

```
/* include the OSC API */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo          = next-value(yourProgressSequence)
```

```
        trExec.cardNo            = "1234567812345678"
        trExec.expDate           = "0305"
        trExec.amount            = 17.25
        trExec.CHName            = "Joe Sakic"
        trExec.CHAddress         = "123 Stanley Cup Place"
        trExec.CHZip             = "11111"
        trExec.enteredBy         = "dan"
        trExec.source            = "oscexample.p"
        trExec.PONo              = "44567"
        trExec.tax               = 1.23
        trExec.shipToZip         = "11112"
        trExec.vatNo             = "GB12345678"
        trExec.commodityCode     = 1234
        trExec.discount          = 1.01
        trExec.shipHandling      = 2.12
        trExec.duty              = 1.04
        trExec.numLines          = 3
        trExec.productCode[1]    = "HSLH-1234"
        trExec.price[1]          = 1.34
        trExec.lineDiscount[1]   = 0.23
        trExec.quantity[1]       = 2
        trExec.description[1]    = "HOCKEY STICK LH"
        trExec.uom[1]            = "EACH"
        trExec.productCode[2]    = "HSKTL-9987"
        trExec.price[2]          = 1.05
        trExec.lineDiscount[2]   = 0.21
        trExec.quantity[2]       = 1
        trExec.description[2]    = "HOCKEY SKATE LACE"
        trExec.uom[2]            = "EACH"
        trExec.productCode[3]    = "HT-90"
        trExec.price[3]          = 3.34
        trExec.lineDiscount[3]   = 0.13
        trExec.quantity[3]       = 75
        trExec.description[3]    = "HOCKEY TAPE"
        trExec.uom[3]            = "FEET"
        trExec.action            = CCA_PURCHASE
        trExec.timeout           = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
    /* error message, return, quit, etc. */
end.

case trExec.authStatus:
```

```
when CCS_RTS   /* <-- NOTICE this is not CCS_AUTHONLY */
then do:
   message "Card approved: " trExec.authDetails.
   /* register order, print receipt, etc. */
end. /* when CCS_AUTHONLY */

when CCS_DECL
then do:
   message "Card declined: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* when CCS_DECL */
```

This example supplied all the PC III fields from the previous example, but it also supplied additional order line data for three order lines. The trExec.numLines field was set to 3, indicating that three order lines' worth of data would be supplied. If we had then supplied only two lines' worth of data, an error would have occurred. If we had supplied five lines' worth of data, line 4 and line 5 would have been ignored (but the transaction would proceed). All of the order line data that we chose to supply as the first line was assigned to the first array element of each line field (e.g. vatNo[1], productCode[1], description[1], etc.), and all of the second line in the second array element (e.g. vatNo[2], productCode[2], etc.).

6.3 CCA_STATUS Example

If you want to check on the status of a transaction at a later time, you may operate in almost the exact same manner as the above examples, except that you will populate the trExec table with the transNo of a previously-executed transaction and use CCA_STATUS:

```
/* include the OSC API in order to use OneStep ChargeSM */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo   = yourPreviousTransNo
       trExec.action    = CCA_STATUS
       trExec.timeout   = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

/* proceed as above, examining trExec.authStatus */
```

6.4 CCA_CAPTURE Example

The CCA_CAPTURE action, described previously, facilitates the settlement of previously-

authorized (CCA_AUTHONLY) transactions. You must supply the trExec.OSCID of the previous transaction, as well as the amount for which you wish to settle. This new amount will then become the final amount of the transaction.

```
/* include the OSC API */
{tri8osc/oscapi}

/* assume yourPreviousOSCID holds the OSCID of the AUTHONLY
// transaction, which was for, say, $20.35.  We will now
// prepare for settlement, but our final amount will be $17.17
// after final adjustments */

create trExec.
assign trExec.transNo   = 1 /* or could use previous transNo
                            // of original transaction */
       trExec.OSCID     = yourPreviousOSCID
       trExec.amount    = 17.17         /* instead of 20.35 */
       trExec.action    = CCA_CAPTURE
       trExec.timeout   = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

/* proceed as above, examining trExec.authStatus, which should
// be CCS_RTS if all goes well */
```

6.5 CCA_QTRANS Example

As described previously, the CCA_QTRANS action is a bit different from the other actions. The QTRANS action facilitates the retrieval of (possibly many) records from the OSC Engine, rather than sending transactions to it. It is particularly useful for reporting. In the following example, we will query the OneStep Charge℠ Engine for all transactions executed between May 21, 2004 and May 25, 2004. Please note that QTRANS and the OSC Engine use PST (PDT during Daylight Saving Time periods) time, so if this example were to adjust into Central Standard Time, a conversion of +2 hours (CST = PST + 0200) would be required for the dates. This example will simply use the dates.

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

create trExec.
assign trExec.action    = CCA_QTRANS
       trExec.timeout   = 100
       trExec.qFromDate = 05/21/2004
       trExec.qToDate   = 05/25/2004.
```

```
run tri8osc/oscengin.p (input-output table trExec).

for each trExec no-lock
   where trExec.action <> CCA_QTRANS:

    display trExec
            with frame tframe overlay centered
            title "Trans Listing"

end. /* for each trExec */
```

Notice that in this case we created a single trExec record, but after the execution of tri8osc/oscengin.p. we anticipate any number of trExec records to exist. We want to skip over the record that was our original query command and view the rest of the records to see what transactions executed between 05/21/2004 and 05/25/2004 (PST/PDT time).

6.6 OneStep Charge Vault Examples

The OSC Vault℠ enables an application to recall and reuse sensitive payment card data without having to store it locally, thereby reducing liability and security needs. Use of the OSC Vault℠ requires a specific subscription, and the functionality must be enabled for your OneStep Charge℠ account.

6.6.1 Vault Example: CCA_STORE

Use a CCA_STORE command to store sensitive data in the OSC Vault℠ for later use. In this example, we'll supply all of the Vault-storable fields, then record the returned vaultID in our database (in the "yourDBTable.vaultID" field).

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo        = 1 /* transNo is not crucial for
                                 // non-payment-transaction
                                 // operations */
       trExec.expDate        = "0310"
       trExec.CHName         = "Joe Sakic"
       trExec.CHAddress      = "123 Stanley Cup Place"
       trExec.CHAddress2     = "Suite 2001"
       trExec.CHCity         = "Denver"
       trExec.CHState        = "CO"
       trExec.CHZip          = "11111"
       trExec.CHPhone        = "800-555-GOAL"
       trExec.CHEmail        = "sakic@halloffame.com"
       trExec.comment        = "Best Wristshot Ever"
       trExec.source         = "oscexample.p"
```

```
        trExec.action        = CCA_STORE
        trExec.timeout       = 20.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
    /* error message, return, quit, etc. */
end.

if trExec.authStatus eq CCS_OK
then do:
    message "Storage successful, vaultID= " trExec.vaultID.
    /* be sure to record the returned vaultID for later use */
    yourDBTable.vaultID = trExec.vaultID.
end. /* if CCS_OK */
else do:
    message "Storage error: " trExec.authDetails.
    /* run new transaction, quit, prompt for action, etc. */
end. /* if not CCS_OK */
```

6.6.2 Vault Example: CCA_STORE (update existing record)

Remember that when using the OSC Vault℠ you must access records using their unique vaultID value. In our first CCA_STORE example we recorded the returned trExec.vaultID value in our sample database table "yourDBTable" in the "vaultID" field (yourDBTable.vaultID). Here we will update the Vault℠ storage for some of the fields.

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

/* let's assume we have the "yourDBTable" record in scope
// that contains the vaultID of the record we want to update--
// our stored vaultID for later use. */

create trExec.
assign trExec.transNo    = 1 /* transNo value isn't crucial */
       trExec.action     = CCA_STORE
       trExec.vaultID    = yourDBTable.vaultID
       /* let's update the expDate, CHAddress and CHAddress2 */
       trExec.expDate    = "0313"
       trExec.CHAddress  = "345 Wristshot Street"
       trExec.CHAddress2 = "Suite 1996"
       trExec.timeout    = 20.
```

```
run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
   /* error message, return, quit, etc. */
end.

if trExec.authStatus eq CCS_OK
then do:
   message "Storage UPDATE successful".
end. /* if CCS_OK */
else do:
   message "Storage error: " trExec.authDetails.
   /* run new transaction, quit, prompt for action, etc. */
end. /* if not CCS_OK */
```

6.6.3 Vault Example: CCA_AUTHONLY (no overridden fields)

Now let's run a payment transaction against a Vault℠ record that we stored in our previous CCA_STORE example. In this example we will supply the minimum cardholder data fields, letting the Vault℠ supply all if the fields we stored previously, without overriding any. We'll run a $17.15 CCA_AUTHONLY transaction. Even though we supply no other fields, the actual transaction that runs in the OSC Engine℠ will include all of the cardholder data we previously stored, because we are supplying the vaultID of the stored record.

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

/* let's assume we have the "yourDBTable" record in scope
// that contains the vaultID of the record we want to update--
// our stored vaultID for later use. */

create trExec.
assign trExec.transNo   = next-value(yourProgressSequence)
       trExec.action    = CCA_AUTHONLY
       trExec.vaultID   = yourDBTable.vaultID
       trExec.amount    = 17.95
       trExec.timeout   = 20.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */
```

```
/* proceed as usual, examining trExec.authStatus */
```

6.6.4 Vault Example: CCA_AUTHONLY (overridden fields)

Now let's run another payment transaction against a Vault℠ record that we stored in our previous CCA_STORE example. But, in this example, we'll supply our vaultID yet we'll also OVERRIDE a couple of values. These overridden values will apply to **this** transaction alone, at run-time.

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

/* let's assume we have the "yourDBTable" record in scope
// that contains the vaultID of the record we want to update--
// our stored vaultID for later use. */

create trExec.
assign trExec.transNo   = next-value(yourProgressSequence)
       trExec.action    = CCA_AUTHONLY
       trExec.vaultID   = yourDBTable.vaultID
       trExec.amount    = 17.95
       /* all our fields will be pulled by the Vault, but let's
       // also override a couple fields */
       trExec.CHName    = "Patrick Roy"
       trExec.CHCity    = "Stillwater"
       trExec.timeout   = 20.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

/* proceed as usual, examining trExec.authStatus */
```

6.6.5 Vault Example: CCA_QVAULT

Let's query the OSC Vault℠ for information about the records we have stored. In this example, we're not filtering by any other parameters (such as cardNo or CHName). Instead, we're just pulling a list of all of our ACTIVE (stored, available) Vault℠ records.

```
/* include the OSC API in order to use OneStep Charge℠ */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo   = 1
```

```
        trExec.action    = CCA_QVAULT
        trExec.timeout   = 20.

run tri8osc/oscengin.p (input-output table trExec).

/* now let's loop through our result set */
for each trExec no-lock
   where trExec.action ne CCA_QVAULT:

    display trExec
            with frame tframe overlay centered
            title "Vault Storage Listing"

end. /* for each trExec */
```

6.6.6 Vault Example: CCA_UNSTORE

Now let's retire our stored data, using an UNSTORE command.

```
/* include the OSC API in order to use OneStep ChargeSM */
{tri8osc/oscapi}

/* let's assume we have the "yourDBTable" record in scope
// that contains the vaultID of the record we want to update--
// our stored vaultID for later use. */

create trExec.
assign trExec.transNo   = 1
       trExec.action    = CCA_UNSTORE
       trExec.vaultID   = yourDBTable.vaultID
       trExec.timeout   = 20.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
    /* error message, return, quit, etc. */
end.

if trExec.authStatus eq CCS_OK
then do:
    message "UN-Storage successful".
end. /* if CCS_OK */
else do:
    message "UN-Storage error: " trExec.authDetails.
```

```
    /* run new transaction, quit, prompt for action, etc. */
end. /* if not CCS_OK */
```

6.7 ACHA_PURCHASE Example

ACH transactions are similar to card transactions in the way they execute. The main differences appear when considering the notion of settlement or “final success” and the fact that funds come right from bank accounts (though the same is true about debit cards).

```
/* include the OSC API */
{tri8osc/oscapi}

create trExec.
assign trExec.transNo   = next-value(yourProgressSequence)
       trExec.routingNo = "12345678”
       trExec.accountNo = "999999"
       trExec.amount    = 17.25
       trExec.action    = ACHA_PURCHASE
       trExec.timeout   = 90.

run tri8osc/oscengin.p (input-output table trExec).

find trExec. /* note that this example assumes only one trExec
             // record exists */

if not available trExec
then do:
    /* error message, return, quit, etc. */
end.

case trExec.authStatus:
when ACHS_ACCEPTED
then do:
    message “ACH transaction Accepted: “ trExec.authDetails.
end. /* when CCS_AUTHONLY */

when ACHS_DECLINED
then do:
    message “ACH transaction rejected: “ trExec.authDetails.
end. /* when CCS_DECL */
```

Once again, keep in mind that ACH transactions and the ACH nationwide infrastructure is much more asynchronous than credit cards. Therefore, merchants may wish to wait to ship products, finalize work orders, etc., until the funds have cleared (or at least 48 hours have passed after the ACHA_PURCHASE execution).

6.8 Test Credit Card Numbers and ACH Accounts

OneStep Charge[SM] supports several test account numbers with predictable responses to help facilitate application testing and OSC evaluation.

The first set of numbers will result in faked "failure" responses without even sending the transaction across the Internet and through the OSC Engine infrastructure. This means that the transactions will not be available for later querying via CCA_QTRANS, CCA_STATUS, and so forth.

Test Card Numbers (Not Sent to OSC Engine)		
Card Number	**Expiration Date**	**Result**
4012888888881	0415	CCS_DECL
4012888888881	0515	CCS_CALL
4012888888881	0615	CCS_ERR

The second list of numbers will send transactions all the way through to the OSC Engine, generating predictable responses. Note that address data is also available on these test cards in order for you to test AVS functionality (optional). Note also that the result depends upon the action you run. For example, if you run a CCA_PURCHASE then the result is CCS_RTS, but if you run a CCA_AUTHONLY then the result is CCS_AUTHONLY.

Test Card Numbers (Sent Through OSC Engine)								
Type	**Card Num**	**Exp**	**Address**	**City**	**ST**	**ZIP**	**CVV**	**Result**
Visa	4111111111111111	1215	123 Test St.	Somewhere	CA	90001	123	CCS_AUTHONLY/RTS
MC	5411111111111115	1215	4000 Main St.	Anytown	AZ	85001	777	CCS_AUTHONLY/RTS
AMEX	341111111111111	1215	12 Colorado Blvd.	Elsewhere	IL	54321	4000	CCS_AUTHONLY/RTS
Disc	6011111111111117	1215	6789 Green Ave.	Nowhere	MA	12345	---	CCS_AUTHONLY/RTS
Diner's	36484444444446	1215	7390 Del Mar Blvd.	Atown	NY	01101	---	CCS_AUTHONLY/RTS
JCB	213122222222221	1215	350 Madison Ave.	Springfield	OH	40000	---	CCS_AUTHONLY/RTS
Visa	4012345678909	1215						CCS_DECL (general)
MC	5555444433332226	1215						CCS_CALL
Visa	4444111144441111	1215						CCS_DECL (card err)

Finally, a similar testing infrastructure exists for ACH.

Test ACH Number (Not Sent to OSC Engine)		
Routing Number	**Account Number**	**Result**
789456124	00000	ACHS_DECLINED

Test ACH Number (Sent Through OSC Engine)		
Routing Number	**Account Number**	**Result**
789456124	55544433221	ACHS_ACCEPTED

7.0 Diagnostics

OneStep Charge[SM] writes logging information based upon the values of OSCConfig.logMethod and OSCConfig.logSpec. It is highly recommended that you use a logging daemon such as the UNIX Syslog to do your logging, in order to handle thousands of virtually simultaneous log messages from many users and sources. To use Syslog, one might set OSCConfig as follows:

```
assign OSCConfig.logMethod = “thru”
       OSCConfig.logSpec   = “/usr/bin/logger -p local3.info”.
```

Alternatively, you may choose to log statically to a file:

```
assign OSCConfig.logMethod = “to”
       OSCConfig.logSpec   = “/var/log/cc.log”.
```

Occasionally, you may want to obtain more detailed information about the processing of a transaction or the operation of OneStep Charge[SM]. This can be done easily by setting the values of OSCConfig.debug flag. Since OneStep Charge[SM] supports runtime configuration overriding via the configuration file (specified in OSCConfig.confFile), you can turn on debug without recompiling anything. Simply set one of these flags in your configuration file, and the next process to load and run OSC will log more verbosely.

8.0 Getting Help

The OneStep Charge[SM] documentation set is, of course, the first source for answers to your OSC questions. In addition, please feel free to consult the OneStep Charge[SM] website at http://onestepcharge.com, where you'll find an FAQ section containing answers to many common questions. Email support can be obtained via psupport@onestepcharge.com. If your OneStep Charge[SM] license entitles you to phone support, this can be obtained at 405.377.3888 (or check the OneStep Charge[SM] website for further/current contact information).