# 4th Quarter® API's to 4D Open

**by Lincoln Stoller, Ph.D.**

Braided Matrix, Inc.
November 2007
v1.41

For information concerning 4th Quarter products contact Braided Matrix, Inc.
Voice (877) 988-1099
Internet: info@4thquarter.com

Braided Matrix, Inc. retains all ownership rights to the 4th Quarter® computer program and other computer programs offered by Braided Matrix (here after collectively called "4th Quarter Software") and their documentation. Use of 4th Quarter Software is governed by your license agreement. The externals written by Braided Matrix and included in 4th Quarter software are confidential trade secrets of Braided Matrix. You may not attempt to decipher or decompile 4th Quarter externals, or knowingly allow others to do so. 4th Quarter Software and its documentation may not be transferred without the prior written consent of Braided Matrix.

The source code, including such items as the file structure, procedures, menus and layouts are the property of Braided Matrix and are protected by copyright. Access to and use of the source code is limited to only those individuals currently licensed by Braided Matrix.

Only such individuals and their employees and consultants who have agreed to the above restrictions may use 4th Quarter Software, and only on the authorized equipment.

Your right to copy 4th Quarter Software and this publication is limited by copyright law. Making copies, adaptations, or compilation works (except copies of 4th Quarter Software for archival purposes) is prohibited by law and constitutes a punishable violation of the law.

BRAIDED MATRIX, INC. PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND. EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL BRAIDED MATRIX BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OF DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF BRAIDED MATRIX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES ARISING FROM ANY DEFECT OR ERROR IN THIS PUBLICATION.

Copyright © 2006 Braided Matrix, Inc. All rights reserved.

The name 4th Quarter is a registered trademark. The 4th Quarter logo, the Braided Matrix, Inc. logo, and the Braided Matrix name are trademarks of Braided Matrix, Inc. The following are registered trademarks of their respective companies:

Apple, Macintosh, LaserWriter
Microsoft, Windows, Windows NT
4th Dimension, 4D Inc., 4D Write, 4D Calc, 4D Draw, and 4D Compiler

# Contents

CHAPTER 1

# API and 4D Open

An overview of 4th Quarter's use of 4D Open to provide other applications with the ability to update 4th Quarter tables.

## *What is 4D Open?*

4D Open is a supporting application that enables one 4D databases to access the data stored in another. The 4D database that is being accessed acts as a server providing data to the 4D Open client.

4D Open also enables the 4D Open client to call methods that are executed on the 4D Open server. 4D Open calls trigger the execution of server-based methods. These run in their own processes and do not return any values to the client.

The 4D Open application differs substantially from the 4D Client/Server. It opens the data base that is managed by 4D Server to be available for reading and updating from any other 4D application. In this manual we'll used the term "client" to mean the 4D Open client, not the 4D Server client. However, the meaning of 4D Server is the same in either case: both 4D Open clients and 4D Server client communicate with the same 4D Server, and these two kinds of communications can go on simultaneously.

Unlike the connection that the 4D Server client establishes with 4D Server, the 4D Open client has less opportunity to communicate and exchange data with the 4D Server. The only interaction between the code in the client and the server comes from:

- Server-based methods. Methods triggered on the server that run asynchronously.

- Triggers: this code, residing on the server, can be configured to run in response to a 4D Open action in almost the same way it runs under the 4D Client/Server environment. The only difference being that error codes cannot be passed back to the 4D Open client.
- Direct Read: where the 4D Open client directly accesses the data stored in the 4D Server database.
- Direct Write: where the 4D Open client writes data directly to the 4D Server's tables largely without any kind of control from the 4D Server application.

All of these methods are support by 4th Quarter except the last. Under no conditions should the 4D Open client write directly to the 4D Server data base. It is a violation of the 4th Quarter User's license agreement to insert or modify data in this fashion.

While data can be read directly, with no modification, by the 4D Open client, modifications should only be done through calls to the 4D Open API's. These are the entry points for the server-based methods mentioned above.

The 4D Open server must be a 4D Server. It cannot be a stand-alone copy of the 4D application. The 4D Open client can either be a 4D Client, a stand-alone 4D application, or another 4D Server application.

## 4th Quarter Support of 4D Open

4th Quarter supports two methods for using 4D Open to access a 4th Quarter database.

- Read Only Access: once a 4D Open connection has been established to 4Q the client application has full read access to the tables in 4th Quarter. A client also has the ability to modify the contents of these tables but 4th Quarter is not designed to support this. In the future it may become possible to physically prevent a 4D Open client from updating the data. At the current time we can only admonish user not to perform any record updating through 4D Open.

  Read access to 4Q tables is handled entirely by the 4D Open client. It requires no special coding in 4th Quarter. All tables are visible to the client. It is the client's responsibility to locate and use this information correctly.

### 4D Open API's

- Method Calling: the 4th Quarter system for updating 4Q tables through 4D Open relies on the4D Open client calling methods on the 4D Open server. The methods called are especially designed for this purpose. This approach constitutes the larger part of 4Q's application program interface (API).

  All of the API method calls share the same structure. They are called with an action parameter and a data parameter. The action parameters control how the called method behaves. The data parameter, which is of type BLOB, provides the data used for updating the records in 4Q.

This manual describes how to use the API methods. This manual organizes methods according to the tables they affect. Separate chapters of this manual are devoted to the use of the calls to update Customer, Vendor, Invoice, Purchase Order, Account, Transaction, and Inventory tables.

#### The Batch Task Module and 4Q Source Code

User's of 4Q's API's do not need access to the 4Q source code. However, if they 4Q source code access, then they will find all 4Q API methods to be part of 4Q's "batch task" module.

All 4Q methods in the batch task module begin with the leading characters "_BT". Most of these methods are an internal part of 4Q's system for handling 4D Open requests. Only one of these methods is actually called by the 4D Open client. This one method is named "_BT4QSubmitTask". This method controls all the functions available to the 4D Open client.

## *Brief Overview of the Batch Task System*

4th Quarter employs its Batch Task system to support 4D Open calls that make changes to the data file.

When the 4D Open client submits a request to update a record, the request is checked for correct syntax, stored in a batch task record. The system then handled the request by processing the batch task record. In this manner the batch task records provide a record of the event, whether the event is successful or not.

For example, when a client requests that a record be deleted, the deletion request is recorded in a batch task record. The batch task record must be processed for the deletion to occur. The administrator can then return to the list of stored batch task records and see when the deletion was requested, what 4D Open client requested it, and what the outcome was.

### 4D Open Client/Server Communication

Because the 4D Open client's request is not necessarily processed successfully by the server, the batch task-based system relies on a certain amount of communication between the two applications. The server must respond to the client's request and it must provide a signal to the client so that the client knows the outcome. This is managed primarily using a shared variable named vBTErrorCode.

The server will only assign a value to the vBTErrorCode if the client is able to connect to the server and issue a command through 4D Open. If the 4D Open connection itself fails, then no server method will have been started and no error code set. The success of the 4D Open connection can be determined by monitoring the error value returned from the 4D Open call itself, that is the 4D Open call ***Execute On Server***.

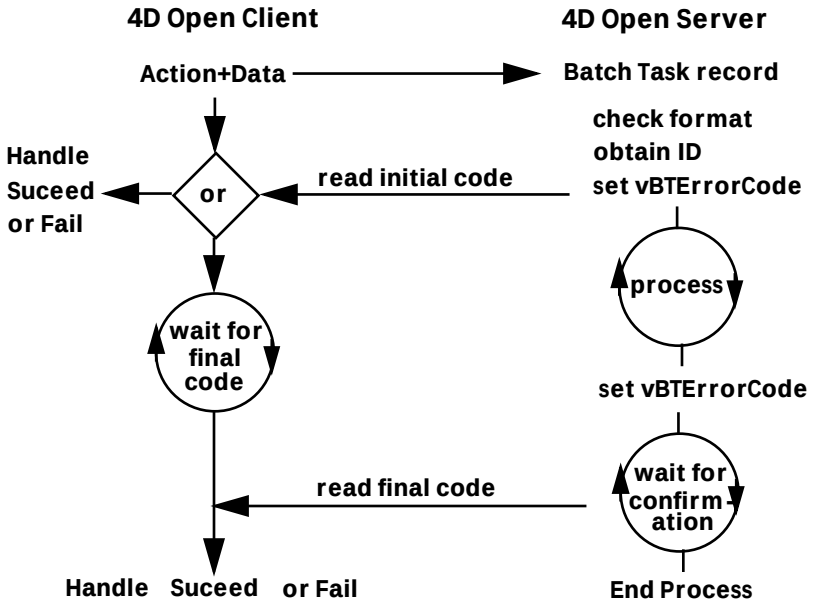
Soon after the 4D Open request has been started a method on the server, the server assigns a text code to the vBTErrorCode text variable. This code contains various parts. The leading part of this code, call ed the Result Code, indicates whether the data submitted by the client was accepted by the server.

The 4D Open client must poll the server to determine the value assigned to this variable. The client must then parse this value to determine the result code. If the result code is a negative number, then the submission was rejected by the server. The supplied data was either incorrectly formatted, or some other condition prevented successful processing.

A positive result code value is assigned when the server-side process accepts the instruction and the data provided to implement the instruction. This result code's value will indicate the primary ID value assigned to the record being created in 4th Quarter. If a record update or deletion is being performed, then the positive returned value corresponds to the ID of the record to be updated or deleted. This is always a positive number.

If the call to the 4D Server is not one that modifies the data file but is rather a request for information, then the result code returned will have the value

zero. The information requested will be passed through one or more other shared variables. This is discussed below.




## vBTErrorCode Variable

The structure of the vBTErrorCode variable is a series of string-type codes concatenated into a single value. These codes are separated by tabs and occur in the following set sequence.

```
vBTErrorCode =
    result code(integer) + Tab +{"TaskID=" + <Task ID> +Tab} +
    current date +Tab + current time
```

The {"TaskID="+<Task ID>+Tab} portion is only included when a Batch Task record is created to process the submitted action. The <Task ID> is the actual ID number assigned to the batch task record created to handle this request. In those cases where a batch task record is not involved, this portion of the code is omitted.

The current date is given in the "short" format and the time in the "HH MM SS" format.

The characters before the first tab comprise the Result Code. A negative value indicates the request was rejected at the start. A positive value means that processing has begun. A positive value does not mean that the request is been fully processed or that it is guaranteed to succeed. To determine the progress of the request sent you can monitor the value that the server assigns to the vBTErrorText variable, discussed next.

The error code variable vBTErrorCode remains blank until the server begins processing the request. Once the requested action has been received, understood by the server, and checked (in those cases where data is supplied), then the server assigns a value to this variable.

The result code assigned is generally the ID of the record being created or modified. If a modification is being performed, then the caller will have supplied an ID in the call and this value will match the value provided. If the call is inserting new data, then the caller will not have provided an ID value and the value given in the result code will be the ID value that will be assigned to the record that is to be created.

The value assigned to the vBTErrorCode variable generally does not change throughout the processing of the submitted task.

## vBTErrorText Variable

The system assigns a value to the vBTErrorText variable at the same time that it assigns a value to the vBTErrorCode variable. The value assigned to the vBTErrorText variable changes over the period of time during which the server attempts to process the request. The value assigned to the vBTErrorText variable will reflect the current processing status of the task.

The value assigned to vBTErrorText explains any errors or problems that are preventing or delaying processing. The 4D Open client can poll the server to determine the value assigned to this variable.

The actual processing of a batch task proceeds just after a new task record is stored in the Batch Task table. This occurs after the system has tested the data supplied in the API call.

In cases where batch task processing is performed the values assigned to the vBTErrorText evolve according to the following sequence.

**TABLE 1.** Structure of the data returned in vBTErrorText

| Form of the assigned value | Description |
|---|---|
| “Batch_Task/0/…” | Data passed initial screens. A batch task record has been created but the job has not been submitted. |
| “Batch_Task/1/…” | The job was submitted once but could not be processed. A second attempt is pending. |
| “Batch_Task/2/…” | The job was submitted twice but could not be processed. A third attempt is pending.<br>The “…” will contain a message describing the reason for the failure. |

**TABLE 1.** Structure of the data returned in vBTErrorText

| Form of the assigned value | Description |
|---|---|
| "OK/<n>/…" | The job was successfully processed on the "n-th" attempt. The "…" provides a text description of the task that processed successfully. |
| "Failed/<n>/" | The job could not be successfully processed within the maximum number of <n> attempts set by the administrator. The job remains in the Batch Task table where it can be re-submitted or deleted by the Administrator. |

The success or failure of a submitted task will depend on:

- Consistency of submitted data: each action imposes different requirements on the submitted data. These requirements range from numerical rules, such as the balancing of transaction components, to the existence of related records, such as the existence of particular accounts.
- Read/Write access to all necessary records. After the server has determined that all the supplied data is consistent and complete it begins to update tables. If some table is locked, then the server will wait for the record to be unlocked. If access cannot be obtained upon the first try, the server will wait a few seconds and try again. The number of attempts that the server will make is set by the 4Q administrator.
- Adherence of the data to processing rules. Some rules apply to the data provided. For example, the rule that available inventory cannot be negative. These conditions cannot be tested until the data file is ready to be modified. It is possible that these rules may not be satisfied at that time.

### Administration

When a task is first received by the server the syntax and consistency of the data are checked. If the data fails these tests the vBTErrorCode is immediately assigned a negative value. The server-side processing of this task halts and no batch task record is created. This condition can be noted by the client monitoring the vErrorCode variable.

In another scenario, the data may be satisfactory but the record being modified may be locked. Here again the server cannot indicate the action has succeeded until after the modification occurs.

The server will enter a loop. At each iteration of the loop it will attempt to gain record access. If the record remains locked, then it waits several seconds and then tries again. If after several tries the server still cannot gain read/write access, then it marks the batch task record as being unsuccessfully. The

value passed back in the vBTErrorText variable will begin with the string "Failed", followed by a slash, and then the number of attempts made.

When the task fails in this manner the task record is marked as "active" and left in the Batch Task table.

The batch task system includes an Administration interface that is accessed by pressing the "**View Batch Task Records**" button on the 4D Open page of the Maintenance screen. This provides a list of all tasks that have been submitted, and the result of their processing.

The Administrator can examine, print, re-run (that is "resubmit the task"), or delete the task entirely without it having had any effect. These options are discussed on page 20. Refer also to the 4D Open section of the Administrator's Manual for additional description.

## *Overview of the Data Entry Methods*

### Calling 4D Open Methods

#### How you setup and use the 4D Open Calls

4th Quarter's API's are based on a 4D Open client's ability to run a specified method on the 4D open server, which in this case is running 4th Quarter.

4D Open not only provides the client with the ability to trigger a server method, but also enables the client to pass parameters to the method. However, it does not provide a direct means for the method to pass information back to the 4D Open client. It does provide an indirect method.

Passing information back to the client is essential. The client must know the outcome of the action in order to know how to proceed with its own actions. Successful actions allow the client to proceed, unsuccessful actions require the client to undo any actions that are otherwise contingent upon the server's success.

#### The structure of the 4D Open calls

All API's communicate to the 4D Server through a single method named *_BT4QSubmitTask*. The 4D Open client sends a message to the 4D Open server to run this method in conjunction with three parameters that are passed at the time the method is called. The _BT4QSubmitTask method is run on the server in its own 4D process.

- Parameter one identifies the 4D Open client. This information is for auditing only and does not affect subsequent processing.

- Parameter two is called the “action code”. The action code tells the _BT4QSubmitTask method what to do. Action codes will generally indicate what table to act upon, and what action to perform.
- Parameter three is a BLOB that is used to pass data to 4D Server. This BLOB will contain different amounts of data formatted in different ways depending on what action is to be taken. All data sent to 4Q is embedded in this parameter.

The options of what can be done on any file are generally limited to:

— Provide a description of the required data and formatting.

— Check supplied data for acceptability in a specified capacity.

— Add a record and perform all related table updates.

— Modify a record and perform all related table updates.

— Delete a record and perform all table related updates.

The action codes and the data structures needed to support these actions are discussed in detail in subsequent chapters.

### The list of all the 4D Open calls

If you pass the _BT4QSubmitTask method that action code “List_All_Actions”, then 4th Quarter will return a list of all the actions codes that are currently being supported by the _BT4QSubmitTask method.

This list will assigned to the vBTErrorText variable and will be in the form of a carriage return delimited sequence of action code values. This list will begin with the action code “List_All_Actions”. At the time the list of all actions has been prepared and assigned to the vBTErrorText variable, the vBTErrorCode variable will be assigned the value “0”.

The client will be able to monitor the vBTErrorCode value on the server in order to know when the list is complete.

The actual list of supported actions will change in future versions. As an example, the current list of supported calls is as follows:

```
List_All_Actions
Account_Description
Account_New_Check
Account_Modify_Check
Account_New
Account_Modify
Account_Delete
Transaction_Description
```

Transaction_New_Check
Transaction_Modify_Check
Transaction_New
Transaction_Modify
Transaction_Delete
Customer_Description
Customer_New_Check
Customer_Modify_Check
Customer_New
Customer_Modify
Customer_Delete
Invoice_Description
Invoice_New_Check
Invoice_Modify_Check
Invoice_New
Invoice_Modify
Invoice_Delete
Batch_Status
Batch_Delete

## How 4th Quarter Responds

### … and how to React to 4Q's Actions

The server method triggered by the 4D Open call communicates back to the 4D Open client through messages assigned to global variables stored within the scope of the server method.

The server method assigns values that can be read by the 4D Open client through 4D Open. The variable values cannot be read directly in the client because their values are not automatically updated in the client when changed on the server.

4th Quarter assigns different values to these variables depending on the actions being taken. Also, different variables are used in different cases. You must refer to the particular API you are using to know what variables to monitor and what messages they may contain. In any case, however, only a small number of variable are used for 4D Open client/server communication.

The global variables used for passing messages are:

vBTErrorCode
vBTErrorText
vOutsideControl
vy1BTText
vy2BTText
vBT1Blob

The first two are variables of text type. vBTErrorCode is assigned a code that identifies whether the call was received successfully or, if it was not, then why not. vBTErrorText provides a description of the status or outcome of processing the task.

The variable vOutsideControl is read by the server process and is used to allow the 4D Open client to terminate server processes prematurely.

The variables vy1BTText and vy2BText are one-dimensional text arrays of varying length. The server uses these arrays to return information in a variety of forms. Each API will use these variables differently.

vBT1Blob is a blob into which the 4D Open server will pack the full contents of the arrays vy1BTText and vy2BText, and any other variables that need to be returned to the client. The variables can be extracted and examined by the 4D Open client. Consult the following descriptions of the calls for details about exactly what variables are packed into the blob.

### Reading the result from the 4D Open server

The 4D Client "spawns" a process on the server in order to perform a particular action. This server-side method cannot return information directly to the client because there is no space of shared variables.

*It is essential to understand that the values assigned by the server to these global variables reside only on the 4D Open server.*

This information cannot be returned, assigned to, or sent back to the 4D Open client by any actions of the server alone. The data can only be retrieved by the 4D Open client who must explicitly request the server to provide the values stored in these variables.

4D Open gives you a way to retrieve the value assigned to a variable in a server process. This is done through the 4D Open call ***OP Get Process Variable***. In this call you specify the target process and the name of the variable that you're interested in. This is discussed further in the following chapters.

### When is the server finished?

The question for the 4D Open client is "When do I request these values which will inform me of the result of my initial request?"

The answer is that you do not know and the server cannot tell you directly. The client must check the variables, and if the task is not done it must wait,

and then check them again. The client must continue doing this until some value is returned or until the client gives up.

What that means in practice is that you must write a Repeat or a While loop in your 4D Open client code. In every iteration of this loop you must retrieve the values of the appropriate variables.

You must also put some a short delay in this loop so that the client doesn't send too many messages to the server in too rapid succession. That would clog up the network and severely degrade performance of the whole system without providing any benefit.

The crucial issue regarding timing is:

- The server process will initialize all of the communication variables to blank text or zero size arrays when the process starts. Values are only assigned after actions have been completed on the server.

The 4D Open client must poll the server. The client must repeatedly requesting the value assigned to these variables until some non-empty value is returned. As soon as some value is obtained, the client can proceed or, if the message indicates the task is incomplete, the client can continue to wait.

### Telling the 4D Open server that you have received the message.

- The 4D Open client informs the server that it has received the message stored in the server's vBTErrorCode variable by erasing the variable's value on the server.

That is, you must make the ***OP Set Process Variable*** 4D Open call to set the value of the error code variable you've just read to a blank.

Once the 4D server completes the task it will wait for this indication that you have received the results of the server process. Only after these message variables have been erased by the client will the server know that its confirmations have been received.

In cases where data is modified there will be generally be some delay between when the server first assigns a value to the vBTErrorCode variable and when it completes the task. The client can either confirm receipt of the information as soon the first message is received that processing has begun, or the client can wait until processing is finished.

Only when the client erases the vBTErrorCode value does the server know that the client has received the message. After the server has completed its task it will wait for an amount of time set by the 4Q administrator for the client to provide this confirmation. If confirmation is provided before the task is

complete, then the server will not wait at all for any additional confirmation once the task completes.

Once the server finishes its task and determines that the client is no longer waiting for confirmation the server process ends. Just like the client the server also has a loop of code that causes it to wait until a confirmation is received that the client has read the vBTErrorCode value. When the server receives this signal it exits its loop and the server process terminates.

Note that once the client indicates to the server that they have read the vBTErrorCode it will not be possible for the client to obtain any further information about the progress of the task on the server.

That is because once the server senses that the client has reset the vBTErrorCode variable to blank, it will no longer wait for the client to receive any further messages. Whatever further actions the server process will perform, it will simply perform those actions and then immediately terminate. And once the server process terminates there is no longer any server process to communicate with and, hence, no way to read the value of the final outcome through the vBTErrorCode variable.

It is actually possible for the client to inquire from the server as to the value of the Batch Task record that was stored as a result of the original request. The outcome of the request will be stored in this Batch Task record. This means of obtaining information at any later time is discussed in the section entitled “Batch Task Status” on page 168.

## Batch Tasks

### Why bother?

Because of the delays inherent in waiting for the server to process the client’s request, and the server waiting for result message to be received, it is imperative that the server handle client requests quickly.

Unfortunately, some of the tasks that are supported through these API’s are neither quick nor predictable in the amount of time required. The time required to fully process these tasks will depend on:

- the size of the 4th Quarter database,
- the memory in the server,
- the amount of user’s on the network,
- the quantity of data being sent by to the server through the API.

In order to avoid bottlenecks and potentially serious issues of contention for shared data, the potentially time consuming API’s employ a system of partial real-time processing followed by batch-controlled final processing.

This means that 4Q's API's perform some initial data and consistency checking and then store the data provided, along with processing instructions, in a Batch Task record. The server returns a reference number to the client that enables the client to access this Batch Task record in the future.

If the data provided in the 4D Open call fails to satisfy the error checking portion of the API, then a message to this effect is assigned to the variables used for communicating with the client. No batch task record is created. If the data passes these tests, then the API's assign a Batch Task record identifier to the communication variable and saves the request in a record in 4Q's Batch Task table.

4th Quarter will immediately attempt to process the information that it has just stored in the Batch Task table. However, due to situations beyond the server's control, processing may be delayed or, in some cases, prevented by conditions that were not tested when the data was originally submitted.

Record locking is the most common reason for a processing delay. It is also possible that between the time the processing request was received, and when the system attempts to process the task, another user may have effected the data so that conditions now preclude successful processing.

For example, in this short span of time the last available item in inventory could be sold, or another user's withdrawal could have wiped out available cash in the checking account. In each of these cases a batch task record will have been created recording the initial attempt. The administrator can view, resubmit, or delete the stored and uncompleted tasks.

The Batch Task records are inherently temporary in nature. They are not considered a primary source of information. Aside from the fact that they are only created as a consequence of 4D Open client actions there are few controls over how long they will remain in the data file. This is discussed further in the section "Batch Task Administration" on page 17.

### How to tell if a submission has been batched?

## Permanent errors

Only 4D Open requests that modify the 4th Quarter datafile are recorded in the Batch Task table. Requests for formatting information, or tests for conditions are not handled through the Batch Task system.

An exception is made when the update request has permanent errors. This could either be due to syntax errors, incorrectly supplied data, or data reference errors. These are errors which cannot be cured simply by waiting.

In this case a Batch Task record is not created. The error is placed in the vBTErrorCode and vBTErrorText variables available to the 4D Open client. No permanent record of the submitted task is kept on the server.

In those cases where a batch task record is created, its ID value is assigned to the vBTErrorCode variable and is placed between the string “TaskID=” and a following Tab character. If the “TaskID=” marker is missing from the vBTErrorCode value, then a batch task record was not created.

## Delaying conditions

If the error is of a temporary nature, such as a locked record, or a condition that may change, such as low inventory or an overdrawn account, then the request will be saved to the Batch Task file.

In these cases an error code and description are in the vBTErrorText variable. 4th Quarter will resubmit the request a specified number of times. If temporary conditions continue to prevent the task from successfully processing after a maximum number of attempts, then the system halts further attempts and saves the task instruction and supplied data in the batch task record.

The batch task records also contain:

- title of the requested action
- both the originally provided BLOB and an “unpacked” display of its contents.
- number of processing attempts
- date, time, and outcome of each attempt
- name of the submitting client
- date and time the tasks were first and last submitted

The entry screen for batch task records is shown in the section of the Administrator’s Manual entitled “Modifying Tasks” on page a.240

## Testing for a batching

In addition to monitoring the vBTErrorCode variable for the string “TaskID=”, there is a second way to tell whether a submitted request is being handled by the Batch Task system. This involves checking the value of the vBTErrorText variable for the embedded word “Batch_Task”.

- If the vBTErrorText variable begins with the string “Batch_Task/”, then the submitted request has been saved in the Batch Task table and has not yet been successfully processed.

  When this occurs the client knows that there were no permanent errors in the information submitted through the API.
- When a batched task is successfully processed the leading string of the vBTErrorText is changed to “OK/”. It is not necessary to wait for this condition as the server will continue with processing in any event. However, if you want to be sure the task completes then you can continue to poll the vBTErrorText variable until this occurs.

### Must the client wait until all processing is complete?

When the 4D Open client receives the Batch Task reference number from the server the client knows two things:

- the server has reviewed the data provided and judged it to be adequate (no permanent errors),
- the server has queued the task for processing.

Depending on the situation, the client can either proceed on the assumption that the task will complete successfully, or the client can continue to wait until the submitted tasks is fully processed.

If you have taken precautions in submitting requests and you are reasonably certain that the task will complete without any data-related error, then it is probably safe to proceed without waiting for the submitted task to be fully processed.

If you want to wait until the batched process has completed, then you must be sure NOT to erase the value in the vBTErrorCode variable. Erasing this value is your signal that you are no longer listening for the server's messages.

The server will continue with its tasks in any case, but if you indicate you are no longer listening, then it will not wait for you to receive any further information. It will terminate immediately upon completing its task.

## Tasks that do not complete

If a task is accepted but does not complete, and that task has been saved as a batch task record, then it is likely the administrator will eventually remedy the situation, remove the problem, and complete processing at a later time. This is contingent upon the actions of the administrator. The client cannot be certain of the final outcome.

If this is not acceptable in your situation, then the only recourse is to delete the unproccessed batch record and reverse whatever actions have been taken on the client. This can be done using the Batch_Delete API discussed further in the section "Stopping and Deleting a Batch Task" on page 169.

### The case of insertions.

In the case where the client is inserting data, it is often necessary for the client to know how to relocate the data within the 4th Quarter data file. This is satisfied by knowing the unique record ID that will be assigned to the inserted record.

For example, if the 4D Open client is inserting a Customer, then the client may need to know the unique ID value that would enable it to locate this customer in the 4Q data file. While the client can assign its own ID to a customer

record that it stores in its own data file, this will not correspond to the ID that 4th Quarter will assign to the customer stored in the 4th Quarter data file.

The 4D Open client may not need this information. The client may be able to locate the new record by searching in the 4Q Customer table according the name or code supplied with the new data. A better tactic is to use the actual Customer ID returned to 4D Open client by the server.

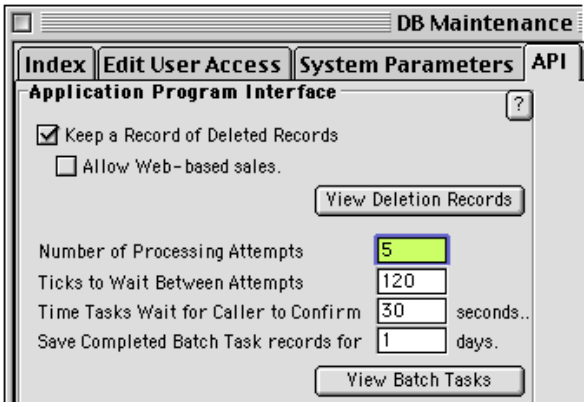
- The reference number that is returned to the client after initial checking is complete is the ID number that will be assigned to the new record when the insertion is performed.

That is to say, in cases where a record is being modified the API returns an ID number that identifies the record that is scheduled to be added, modified, or deleted.

## *Batch Task Administration*

This section provides an overview of the Administrator functions relating to Batch Task records and the control of Record Deletion logging. Addition details can be found on page a.236 of the Administrator's Manual.

Several settings appear on the API page of the Maintenance screen.




### Track Deleted Records

The "Keep a Record of Deleted Record" and the **View Deletion Records** button are used to track record deletions so that remote databases can synchronize their contents with the contents of 4Q. This is discussed further in chapter 2, "Synchronizing Data Files" on page 25.

### Allow Web-based Sales

The “Allow Web-based sales” check box instructs the system to create a new client type that’s named “web client”, and with this new type the system creates the usual accounts associated with a client type. This provides a means of segregating those clients created through the API from those created from within 4Q itself.

### Number of Processing Attempts

This is the number of times that 4Q will resubmit a batch task that is delayed due to temporary conditions, such as a locked record. 4Q will wait a few seconds between each attempt before resubmitting the batch task.

The number of attempts that will be needed to ensure that a task runs to completion will depend on your application and how much data contention it generates. Set this value higher to ensure complete processing of each task. Set it lower to prevent backed up tasks from overwhelming the system.

Tasks that don’t complete successfully are saved on the system indefinitely. Tasks that do complete successfully are only saved for the number of days indicated in the “Save Completed Task Records for” field, described below.

### Ticks to Wait Between Attempts

This is the number of ticks (1 tick = 1/60 second) the system will wait after a unsuccessful attempt has been made to process at batch task record, before the same request is resubmitted. Batch tasks will fail either because of some multi-user issue having to do with the unavailability of shared data, or because of an actual insufficiency of some resource that is needed to complete the action.

In the first case the problem may be resolved by a brief pause and then resubmitting the request. In the second case the user will need to examine the error code returned after the batch process has failed, remedy the situation, and then resubmit the task.

### Time Tasks Wait for Caller to Confirm

This is the time that the system will wait for confirmation from the caller after it has either successfully processed the batch, or reached the maximum number of allowed processing attempts.

Once the batch task process has reached either of these points it has nothing further to do except wait for the caller to confirm receipt of the result code placed in the vBTErrorCode variable.

The caller confirms receipt of this message by monitoring the variable and resetting it to a blank value after the server has set it to a nonblank value, as discussed above.

Since the server sets the vBTErrorCode variable when it starts processing the batch task, the remote system may have already received the message before the server completes the task. If the caller reset the vBTErrorCode variable

before the server completes the task, then the server will not wait for any further confirmation once it finishes the task. In this case the time you set in the “Time tasks wait...” field has no effect.

On the other hand, if you have not reset the vBTErrorCode variable by the time the batch task completes its processing, then the server will wait for confirmation for the time indicated by the “Time tasks wait...”field.

If the caller confirms receipt of the message before this time limit, then the batch task process finishes and both the variable and the task process become undefined. If the user does not confirm receipt of the message an the indicated time limit is reached, then the batch task finishes without confirmation.

The server does not perform any further submissions of incomplete tasks during the time it’s waiting for confirmation.

### Days to Save Batch Task Records

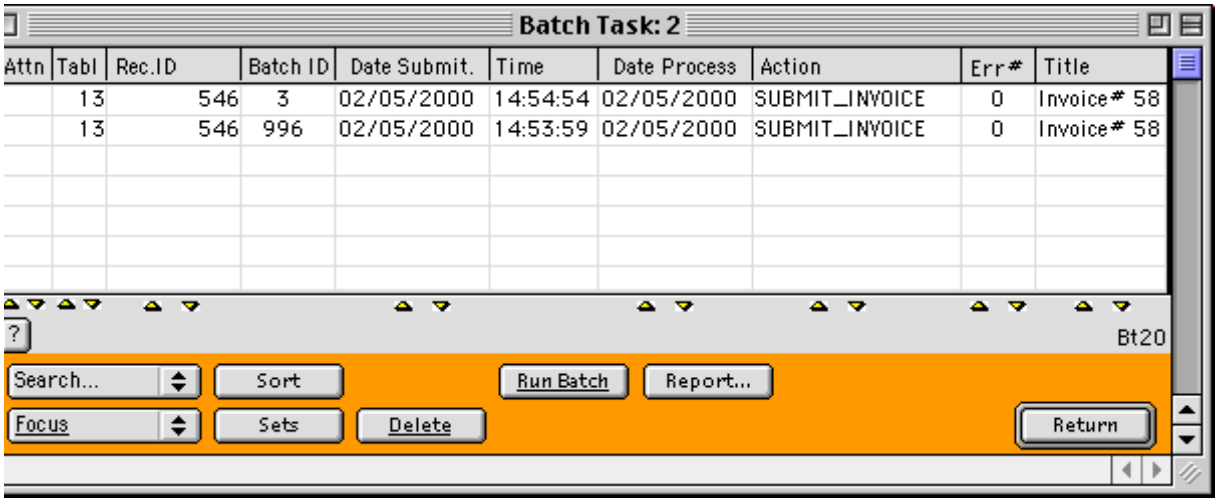
Batch Task records associated with successful projects are retained for a limited time. After this time they are automatically reused. The amount of time successful Batch Task records are kept is determined by the value assigned to the field labeled "Save Completed Batch Task records for <…> days".

The Save Task Records for value should be set for 5 to 30 days. Set higher values for a longer audit trail.

Batch Task records are never automatically deleted, but they are only overwritten with new Batch Task data as it is submitted. This means that even if you assign the value zero for the number of days that task records are to be saved, the records are still saved and may remain in the system for some time before they are overwritten by a new request.

### Viewing Batch Task Records

Pressing the **Viewing Batch Task** button opens the Batch Task list screen.




This screen provides access to all of the system's Batch Task records, those processed and still waiting to be processed.

Choose the Select All item from the Search popup menu to display all tasks in the system.

## *The Pros and Cons of 4D Open*

Access to 4Q through 4D Open offers the following benefits:

- Access to 4Q data without any risk to the integrity of the 4Q application.
- The Server handles all updating through the Batch Task system. This relieves the client of the responsibility and provides Administrative oversight.
- Data files are kept separate, and access to the data in the applications is easily controlled.
- The opportunity to have the 4D Open client and server reside on opposite sites of a network firewall. In this case 4D Open is used as a local protocol providing limited and controlled access to server data that is invisible to outside network users.
- Since client requests are processed entirely by the server, the client is not burdened by the network traffic or processing load that affects the server's operation.

- Updating of either or both of the two applications is simple as long as they have not been customized. One only needs to replace the previous version on your server with a new version. There is need to manually update the data.

At the same time, 4D Open imposes severe limitations on a programmers ability to integrate the client and server applications.

- The client should not exercise their own rules in updating 4Q's tables.
- The server's response to client requests is not synchronous. This makes it difficult for the client to react to the effects that its request has upon the server data.
- The server does not share any variables with the client, cannot pass or receive messages from the client (except for the initial submissions), and neither application can use pointers to objects that reside in the other.
- The client cannot respond to server-side triggers.
- The server has no access to the table data or variable space on the client.

These hardware limitations make it difficult to create a real-time, integrated system based on 4D databases communicating with 4D Open. Instead, one should use 4D Open for those tasks for which it is most suited.

Avoid using 4D Open in situations where tight control, synchronization, or rule-based integration is required. In those cases a direct integration of the structures would be better.

## 4Q's Data Entry API's

Updating tables in 4th Quarter using 4D Open is done by making calls to 4th Quarter's Application Program Interface (API). This interface consists of one or more methods that take various parameters depending on the task you want to perform.

In most cases the 4D Open client will call a method named _BT4QSubmitTask. When called through 4D Open the 4D Server will execute this method in a server process.

The _BT4QSubmitTask method takes the following parameters.

$1 (Text) = Name of the caller's computer, application, and method.
$2 (Text) = Action code.
$3 (Blob) = Data in specific format that depends on what action is requested.

After _BT4QSubmitTask finishes executing its task, it assigns values to certain system variables. The 4D Open client can read these variables through 4D Open in order to learn of the result of the submitted operation. The variables that _BT4QSubmitTask method uses for communication are:

| | |
|---|---|
| vBTErrorCode | Text |
| vBTErrorText | Text |
| vOutsideControl | Text |
| vy1BTText | Text array |
| vy2BTText | Text array |

After reading variable values the 4D Open client should reset the vBTErrorCode variable's value to an empty string. This action indicates to the server that the client has received the essential processing results. The server process will wait for this signal and, once it has been perceived, the server process running the _BT4QSubmitTask method will terminate.

### Asynchronous Communication

The communication between the two applications using 4D Open is asynchronous. That is, once the information has been passed from the 4D Open client to 4th Quarter both the client and 4th Quarter continue executing without waiting for, or communication with the other application.

However, the nature of the tasks being performed requires additional communications between the client and the server. There are two reasons why this is needed:

- There may be something wrong with the client's call to the server.

  — the syntax of the call may be incorrect,

  — the data passed may be unacceptable, or

  — conditions may exist on the server that prevent the task from completing.
- The client usually needs a reference number when it creates new records on the server. The client will use this number to establish a relation between the information just submitted and stored on the client, and the consequential actions or data that are stored on the server.

  For example, if the client submits an invoice, then the client will want to know the ID of the invoice as it is stored on the server.

4th Quarter uses the previously mentioned variables as "containers" for the passing of messages from the server back to the client. This means of message passing is similar to how a message can be passed between to 4D processes using 4D global process variables.

This method relies on the client and the server agreeing on what variable will be used to pass a message, and it requires that the client poll the server for a value that's been assigned to this variable.

In practice, the client must wait for the server to respond through a value that the server assigns to a message variable. The client code must enter a loop that periodically reads the value of the message variable. If the value is blank, then the client continues to wait until it is not blank.

In turn, the server must complete its task expeditiously and return a message. Any delays on the server side will impact client performance.

Inter-application message passing between the client and the server must be disciplined and reliable. For every task that the client can submit, the nature of the responses that can be returned by the server must be determined in advance.

## *Testing 4Q's API's*

The 4Q API's are designed to be called from outside 4th Quarter through an inter-application protocol such as 4D Open. However, for the purposes of testing, they can also be called from within 4th Quarter using a method designed especially for this purpose.

This does not fully replicate the situation that ensues when the API's run as a server process. However, when called internally the API's will perform all the actions they would if called remotely. This enables you to test the packing and unpacking of the parameters as well as some of the synchronization issues crucial to the success of the API calls.

These test calls executed with the 4Q application itself are easier to trace. Alert messages encountered either by the calling or called method will be displayed on-screen. The test method also can write a log file that notes the reading and writing to the common vBTErrorCode and vBTErrorText variables.

For details on how to use this testing method refer to the section "The _UTILTestBTSubmit Method" on page 171.

CHAPTER 2

# Synchronizing Data Files

A review of the methods available in 4th Quarter to synchronize tables shared by 4th Quarter and a remote client application.

## What is Does Synchronizing Mean?

4th Quarter and a related, remote application may need to share data. These applications may also need to maintain this common data in separate data files because of the difficulty of storing data in one application's datafile.

Cases where this may be advisable occur when applications are designed and maintained independently, yet both need control of, and access to common data. In these cases it is important that the information held in common be consistent between the two applications.

Creating two data files that store some common information provides this independence, but it involves additional overhead in order to maintain separate and consistent sets of data. The possibility that the common data may fail to agree due to changes made on one side or the other must be controlled.

Controlling the data in separate application requires a means of:

- communication between the applications,
- discovery of inconsistencies, and
- resolution of inconsistencies.

This can be satisfied using:

- a communication protocol, such as 4D Open,
- the API's built into 4th Quarter for this purpose,
- rules for how and when to locate discrepancies, and

- a strategy on the client side for keeping the client data consistent with that master data stored in the 4th Quarter data file.

### Synchronizing is the Client's Responsibility

The client application that must handle the synchronizing of data. This is true whether the data handled by 4D Server is accessed through the use of 4D Open or some other means. The reasons for this are as follows.

- Client connections are one-way connections. This means that only the client application can see and exercise some control over *both* data files.
- 4th Quarter's data file is the master data file. This is certainly true for accounting data and is generally true for other data that is related to the accounting data.
- The 4th Quarter application does not have "hooks" that would make it easy to install synchronizing rules or methods into the application. This means that should you make 4th Quarter data not primary, then you would need to modify the 4th Quarter application, thereby creating an new version of 4th Quarter that would not be immediately compatible with future versions.

An exception to these rules would occur in areas of 4th Quarter that you were already customizing and which did not involve accounting data. Since these customized areas depend on custom programming and logic, they could be created with those elements necessary to behave as secondary data.

In this chapter we will only deal with the case where 4th Quarter stores primary data. The extension of this to the customized case where the primary data is in the remote data file can be handled by adapting the logic discussed below.

## Synchronization Rules

Various different mechanisms can be used to maintain a correspondence between the data on the client application and the 4D Server application.

Shared data can, but does not need to be identical. Data that is shared between data files can differ in form and still be considered equivalent. And if the shared data is equivalent to any degree, then it will need to have some method to ensure that it is synchronized.

### Customer and Account Example

It is possible to make use of 4th Quarter's accounts to track customer balances without keeping record of the customers themselves. That is, you can build an application in which the customer records are stored in one database, and the customer account records are stored in 4th Quarter.

A mechanism to preserve the correlation of customer records with their associated accounts records needs to be shared between the two databases.

You would probably share the customer's name and balance in both data files. You would need to ensure that changes to these shared values in one data file were propagated to the other data file in a timely manner.

Since it is the client application's responsibility to monitor and enforce changes necessary to keep the data synchronized, the client needs a means of noticing changes in the server's data file.

The client also needs a means of checking data on the server against the data on the client to determine if the changes are significant.

The client then needs to resynchronize the different data using one data file as the master.

## *How to Synchronize*

### Synchronization Tools

The two structures in the 4th Quarter data file useful for synchronizing are the modification times assigned to records, and the deletion log that's maintained for certain tables.

#### Modification time

Whenever a user modifies a 4th Quarter record, 4th Quarter stamps the record with a modification date and time. Not all tables support this feature. It

is implemented only for those tables considered most likely candidates for synchronization as shown in Table 2.

**TABLE 2.** Tables Supporting Last Modified Date & Time

| Field Names | Tables | Modification Conditions |
|---|---|---|
| LastModifiedDate<br>LastModifiedTime | Account | When modified through the record entry screen or using Apply To command. Not restamped as a consequence of transactions or posting. |
| | GL_Account | |
| | Transaction | When modified through one of the record entry screens or using Apply To command. Not restamped when allocated or posted. |
| | Customer | When modified through the record entry screens or using Apply To command. |
| | Vendor | |
| | Inventory | When modified through the record entry screens. Not affected by shipping or receiving. |
| | vInvoice | |
| | vPurchaseOrder | |

All the tables in the listed Table 2 have the fields named “LastModifiedDate” and “LastModifiedTime”. These fields are automatically updated whenever a modification is made to a record in the corresponding file. This enables the client application to locate those records that have been modified after any fixed date.

As noted in the column labeled “Modification Conditions” not all changes to the records cause the system to reset the record’s modification date and time. In general, only changes to the primary data stored with the record cause the record to be restamped. Changes to secondary, or non-normalized data, such as subtotals and information copied form other tables, do not cause the system to change the record’s last date and time of modification.

### Update all records modified after last synchronization.

If the remote user is performing synchronization operations by copying data from the remote server, then it should also be maintaining some record of the date and time when this was last performed.

That is, if the remote user runs some batch process that reestablishes the equivalence of the remote user’s data with the data stored in 4th Quarter, then the remote user should also make a permanent record of when this is last performed.

Knowing when an update was last performed enables the client to search the server for records in the selected tables that have modification date and times that subsequent to the time the last update was performed. Without this information the client would have to review all the records in the two data files every time they wanted to reestablish equivalence between the two data files.

Using the date and time of the last update enables the client to examine on a limited number of most recent changes made to the server data.

After the more recently modified records have been located the client needs to create a list of the modification dates and times assigned to these records. This information can be returned to the client in a set of arrays containing the following information.

Array #1: Record ID,

Array #2: Modification date

Array #3: Modification time

This would be done by establishing a bind between the ID, date and time fields in the targeted tables on the server, and arrays that are local to the client.

Once the client has obtained this information, the client then look's up the same records in its own data file that match the ID's of the returned records. (This assumes that you have stored the ID used on the server to identify these records.)

There are four outcomes that can result from a comparison of the corresponding records in the two data files:

- Client data file has no record with the corresponding ID.

  — The record must be created in the client's secondary data file.
- Client record is found with modification date and time that match that in the primary data file.

  — The record does not need to be updated.
- Client record is found with a combined modification date and time that is *earlier* than those in the primary data file.

  — The secondary data file needs to be updated.
- Client record is found and the combined modification date and time are *later* than those in the primary.

  — The *primary* data file needs to be updated.

To update the secondary data file you must locate the record and bring its data across using the tools available in the communication channel (e.g. 4D Open).

To update the primary data file you can use the modification methods supported in 4Q's API, as described in this manual.

## Deletion log

Normally, the deletion of a record leaves no trace in the data file from which it has been deleted. Without some additional reference, such as an archive, one would not be able to know what records have been removed.

The lack of any archive would present a problem for keeping data files synchronized. 4th Quarter's deletion log provides such an archive.

The deletion log is a separate table of records maintained solely to record those records deleted form certain files. The one deletion log table, named "DeletedRecords" is used to track deletions in all of the tables that are monitored.

Only certain tables are monitored, namely those tables that are candidates for mirroring in a client data file. This is the same list of tables for which modification dates and times are kept as shown in Table 2.

The Deletion table stores a minimum of information identifying only

- record ID,
- table from which it was deleted,
- user who deleted the record,
- date of deletion.
- time of deletion.

The client application can determine what records have been deleted in any of the monitored tables, within any time period, by searching the Deletion Log table. Using this information the client application can delete any records in its own data file that remain after the corresponding records have been deleted from the 4th Quarter data file.

## Synchronization Time-Frame

We are calling the frequency at which the client tests for equivalence between the data stored in the two files is called the *synchronization time-frame*. This time-frame can be anything from continuous to occasional, it can differ as a function of the circumstance or the kinds of test performed.

For example:

- A thorough test for changes might be run once each day, while less intrusive tests might be run periodically throughout the day.

- Changes to the client's data file might be send immediately to the server, while changes in the server's data might only be tested at the end of the day.

There is no general rule governing how frequently the client data file should be brought into coincidence with the server data file. Deciding on a synchronization time-frame is specific to the needs of individual applications.

### Example of Batch Synchronization

The 4D Open client can send inserts, updates, and deletes to the 4D Open server as soon as they take place.

The 4D Open server can be queried automatically on a daily basis for the purpose of synchronizing the client's data file with changes made to the 4Q data file by mean other than client updates.

## Means of Resynchronizing

There are two parts to synchronizing.

- Passing updates made to the remote (client) data file back to the master, (server) data file.
- Passing updates made to the master back to the remote data file.

In the first case these are generally handled in real-time, in the second case they are handled in batch-mode.

### Passing updates to the master data file

Insert, update, and delete operations can be performed by the client application using 4th Quarter's API's. You *do not* need to establish "binds" between the tables on the client and tables on the server for this operation.

Tasks submitted to the server by the client begin processing immediately. In the case of deletions the updates are completed before any error code is set for the client. In these cases the client waits for the server to perform the requested task before proceeding.

In the case of inserts and updates the task is handled by 4Q's Batch Task system. The task is started immediately after the task i s created, but the server returns an preliminary error code before the task is completed. This Batch Task system enables the client application to execute synchronously or asychronously with the operation of the server.

**Synchronous** If the client wants the security of knowing the submitted task has been successfully completed before proceeding, then it can obtain this by waiting for 4th Quarter's Batch Task system to finish processing.

**Asynchronous** If the client does not need this level of control, or if the client is in greater need of rapid response, then the client can proceeded after the short wait required for the submission to be checked for basic content and syntax. In this case, after a brief wait, the client and the server proceed asynchronously.

### Retrieving updates to the client data file

Data is read by the client from the server's data file through direct access to the 4th Quarter tables. You *do* need to establish "binds" between the tables on the client and tables on the server in order to perform this operation.

If you're using 4D Open, then access to tables is through 4D Open calls that use binds made between the fields on the server and fields or variables on the client. Similar binding methods are supported for other communication protocols.

4D Open enables the client to locate records on the server using the commands like OP Multi query, OP Single query, and others. Refer to the 4D Open documentation for details on using these commands.

## Deletions

Handling the deletion of records is a special case in the general problem of synchronizing data files. Deletion is more than just a particular kind of record modification because the record's data is entirely wiped out when the record is deleted. This loss of information requires a special method for managing deletions.

We are concerned with two kinds of deletions:

- deletions on the server that need to be updated on the client,
- deletions on the client that need to be updated on the server.

### Deletions on the server

Server-side deletions are tracked by 4th Quarter's Record Deletion log. Record deletion logging must be turned on in the Administrator's Maintenance screen, on the API page.

Once Deletion Logging is turned on, 4th Quarter keeps a record of what records were deleted, and when they were deleted, in a special Deletion Log table. You can use this information in either of two ways:

**Search the Deletion Log for particular records.** Use this method when you are managing a small number of records, or a particular set of records. This method is generally limited to system tables or tables that are rarely modified.

**Search the Deletion Log for entries within a date range.** Use this method to track changes to a non-system table, or a table that contains many records.

To use this method effectively you must keep track of when you last searched for deleted records. Knowing that you can search for deleted records from that date forward. The records found will correspond to items deleted since the query was last run.

There are two kinds of records that can be found.

- Records that exist on the client-side.

  These records need to be deleted from the client side.
- Records that do not exist on the client-side.

  This would be the case for records that were created and deleted wholly on the server side.

### Deletions on the client

Deletions on the client are easier to track since you have the ability to control and to record the event in whatever manner is most convenient. However there is the serious problem that since the server data file is considered the master data file, the server application is also the final arbiter of whether or not any record can be deleted. This means that while it is easy for you to control a client-side deletion, you must take extra care to ensure that the deletion will be allowed by the server.

The easiest way to determine whether or not the server will allow a deletion is to perform the deletion from the server before performing the deletion from the client. That is, use the 4th Quarter API for record deletion before deleting the record from the client data file.

This can be done by calling the 4Q API to delete the record from the server's data file. If the deletion succeeds, the proceed with the deletion on the client side. If the deletion fails then cancel the deletion on the client side.

In the vent that changes need to be made on the client-side first, then start a 4D Transaction before making any changes to the client-side date. Proceed to modify the information on the client-side and then use the 4Q API to delete the data on the server. If the server deletion succeeds, then validate the transaction. If the server deletion fails, then roll back the transaction.

### Preventing Client-side Deletions

Another approach to managing deletions is to prevent them to be performed from the client side. This means than any deletions that need to be performed must be done through 4th Quarter on the server data file.

In this approach you do need to update the server's data file in response to client-side inserts and updates but not deletions. You only need to update the client data file to reflect deletions made on the server side.

Deletion management still needs to be done on the client-side, but you gain greater control. This approach also requires less programming.

## *Administration of Record Deletion Logs*

### Enabling Deletion Logging

Deletion logging must be turned on through the API page of the administrator's Maintenance screen. This only requires the check box on that page be set.

Checking and unchecking this box affects whether or not future deletions are logged. It does not effect the logs created for past deletions.

### Reviewing Deletion Log

Deletion Log list screen, opened from the API page of the Maintenance screen, shows all deletion log records. From here you can search to create reports of records deleted for specific tables, within specific data ranges.

Use the Sort button and the Report button to create reports covering the currently selected deletion log records listed in the displayed order.

Use the Delete button to remove old deletion log records. Deletion log records are not automatically deleted by the system. They are only removed by manual direction of the administrator.

In general, the deletion log table requires no maintenance. If there are many records in the table then old records can be purged to improve the speed with which the remote client can locate and transfer information from the server.

The total number of records in the table can be determined by searching for all records and viewing the number found in the title bar at the top of the screen.

CHAPTER 3

# Account Data Entry Methods

Updating the account table using the 4th Quarter API's and 4D Open.

4th Quarter supports account-related tasks through the _BT4QSubmitTask method.

Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method. Each task passes a different "action" parameter and, according to the task, a different BLOB of data.

**TABLE 3.** Account Table Operations

| Action Parameter | Description |
|---|---|
| Account_Description | Returning a description of the account data format. |
| Account_New_Check | Checking the data supplied for a new account. |
| Account_Modify_Check | Checking the data supplied for a modified account. |
| Account_New | Inserting a new account. |
| Account_Modify | Modifying an existing account. |
| Account_Delete | Deleting an account. |

Each task will different information back from the server. Details are described below.

## Account Data Format

### Returning a description of the account data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new account. This information is passed back in the text of the variable vBTErrorText.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “ACCOUNT_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $vMethod; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
        is opened.
`Note: pass only the names of process variables, not local variables,
        when providing variable references in 4D Open calls.
vIdentity:=Application File+ ";" + Current User +
        <name of current method>
vAction:= "Account_Description"
$vMethod:="_BT4QSubmitTask"
$ProcessName:="TestAccount"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
v4QLockoutFlag:=""
$4QIsLocked:=False
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
        $MyProcessID; "<>vSYDB_LockoutFlag";
        "v4QLockoutFlag")

If($4DOpenErrCode = 0)
    $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
        “<>vSYDB_LockoutFlag”; vSemaphoreState)
    $4QIsLocked:=vSemaphoreState
    $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
```

```
                    "<>vSYDB_LockoutFlag")
End if

Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
            $vMethod; $Stack128K;$ProcessName;
            $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)
                              `wait 30 seconds max.
      $TicksToWait:=20        `delay between 4D Open calls to
                              `prevent flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                        $MyProcessID;
                              "vBTErrorText"; "vMessage")

      While ((vBTErrorCode = "") &
                              (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorCode";
                  "vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorCode";
                  "vBlank")
      End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
```

```
        End if

        Case of
        :($BTErrorNumber<0) ` Server has returned an error.
              ` Take appropriate response.
        :(vBTErrorCode="")
              ` Timed-out waiting for a response.
              ` Process error here.
        Else
              $AccountDataSequence:=vMessage
              ` Task completed successfully.
              ` Display account data text.
        End case
  End case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and times values are not provided.
- Empty string: processing has not begun.
- Account ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the account data that 4th Quarter expects to receive as elements in a text array when you submit a new account through a 4D Open call. The same information is also available in the vy1BTText array. The contents of this array have also been stuffed into the blob vBT1Blob which can be read from the 4D Open client.

The structure of the vBTErrorText message is as follows:

**TABLE 4.** Structure of the Account Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required text array. |
| 2 | A header identifying the following information as pertaining to account data. |
| Starting at 3 | One line descriptions of each of the elements that the account text array, passed in the BLOB variable, is expected to contain. |
| This array is stuffed in the blob vBT1Blob. | |

The characters in vBTErrorText before the first carriage return give the size of the array that is expected. This value can be extracted by finding the position of the first carriage return, extracting the characters lying before it, and converting them to a number.

A heading line follows. After that there are one-line descriptions of each of the account array elements. These descriptions are placed in carriage return delimited format. That is, each element of the account array is described using a separate line in the vBTErrorText variable.

This same information that is assigned to vBTErrorText is also placed in the text array vy1BTText that is stuffed in the blob v1BTBlob. The array can be extracted and its size determined using the Size of Array command.

All of the account data will need to be placed in a text array before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text array must then be packed into a BLOB. This BLOB is passed as $3 when the call is made to create the account. Refer to the following section “Inserting a new account” on page 46 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

```
28
28 element account text array
Account ID
related GL Account ID
```

Use GL Account defaults key(1,-1)
Account Number
Account Suffix
Account Name
ID of related Equity Account
Related File Number
Related Record ID
Related Record Name
External Account Number
Description
Reference (name on check)
Attention
System Action
Address Text
UseLinkedAddress(1,-1)
Active Key(1,-1)
Access Key(1,-1)
Finance Rate Per Month
Days Grace
Monitor Key(1,-1)
Debit Caution Level
Debit Notify Level
Credit Caution Level
Credit Notify Level
Next Check Number
Autoallocation Key(1,2,3)

Execute the “Account_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

# Checking Data

## Checking the data supplied for a new account

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new account record described in the BLOB satisfies 4Q's account entry requirements.

This checking requires:

- The account information be properly formatted in the BLOB, and
- The data supplied is consistent with 4Q's account entry requirements, as are described below.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- It *does not* determine whether or not the account record can be immediately created.
- It *does not* lock any records in anticipation for the actual submission of new account information.
- It *only* checks the data for entry as a new account. It does not check the information passed for validity if used to modify or delete an account. Those actions only require the existence of the record and the specification of a record ID.

Getting confirmation from this task does not guarantee that the account can be created, modified, or deleted immediately.

Other concurrent processes may have access to needed sequence number counters at the time when the account data is submitted.

If such record locks occur when the account is submitted the server will cache the account data in a batch task record until updating can be done.

To start this task to check the data provided for the creation of a new account call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "ACCOUNT_NEW_CHECK"
$3= Full account data packed in the BLOB (formatted as described below).

To start this task to check the data provided for the modification of an existing account pass the following parameters.

$1= Application, user, and method name in a text string.

$2= “ACCOUNT_MODIFY_CHECK”
$3= Full account data packed in the BLOB.

The data passed in the BLOB must first be placed in a text array of the correct size. This array data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. Although not shown, all the intermediate array elements need to be assigned.

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyAccountArray;46)
$MyAccountArray{1}:="0"
  ...
$MyAccountArray{46}:="TheAddress@TheDomain.com"
VARIABLE TO BLOB($MyAccountArray; $DataBlob)
```

The current account data structure is given in Table 5. The table also provides notes regarding the use or formatting of the data that must be supplied.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases where one or another item is required. This is described further in the “Description” column.

Not all fields in the account table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

A complete description of the account fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the account data structure expected in the current version of 4th Quarter that you are communicating with you should execute the “Account_Description” call.

**TABLE 5.** Account Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 1 | X[0] | Account ID | [0]Not used for a new account. Must be the actual > 0 value for an existing account. |

**TABLE 5.** Account Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 2 | X[1] | GL Account ID | [1]Required for a new account. Not used for an existing account. |
| 3 | | Use GL Account Defaults | key value, 1=yes, any other value=no.<br>If GL account defaults are used, then all GL account defaults are applied to the new or modified account without regard to what values are passed in the API for the corresponding fields. |
| 4 | | Account Number | Does not include the GL account number |
| 5 | | Account Suffix | |
| 6 | | Account Name | |
| 7 | | Equity Account ID | |
| 8 | X[2] | Related File Number | File and record numbers should either both supplied, or none supplied. Record name is for display only. It is not updated automatically and is optional. |
| 9 | | Related Record ID | |
| 10 | | Related Record Name | |
| 11 | | External Account Number | User defined auxiliary reference |
| 12 | | Description | |
| 13 | | Reference Name | Name that's used as Payee/Payer when creating receipts or disbursements. |
| 14 | | Attention | |
| 15 | | System Action | unused |
| 16 | | Address Text | |
| 17 | | Use Related Record Address | key value, 1=yes, 0=no<br>Address taken from Address record linked to the Related File and Record numbers. The address must exist. |
| 18 | | Active | key value, 1=yes, -1=no |
| 19 | | Access | key value, 1=general, -1=restricted |
| 20 | | Finance %/Month | Expressed in the form of a percent. |
| 21 | | Days Grace | |

**TABLE 5.** Account Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 22 | | Monitor Balance | key value, 1=monitor, 0=don't monitor<br>Monitoring done when transactions are specified. |
| 23 | | Debit Caution | |
| 24 | | Debit Notify | |
| 25 | | Credit Caution | |
| 26 | | Credit Notify | |
| 27 | | Next Check Number | |
| 28 | | Automatic Autoallocation | key value, 1=yes, 0=no<br>Determines allocation for AR and AP. |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.

**TABLE 6.** Account Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -118 | Specified Account does not exist: <ID=…> |
| -120 | No account ID. Only required when modifying an account. |
| -125 | No GL account ID. |
| -135 | Specified GL Account does not exist: <ID=…> |
| -140 | Duplicate account number for indicated GL account: <Nº=…> |
| -145 | Equity account does not exist. <ID=…> |

**TABLE 6.** Account Data Value Error

| Error Code | Description |
|---|---|
| -150 | The account has no name. |
| -215 | Account ID could not be obtained or could not be assigned. |
| -230 | Account could not be deleted. |
| -235 | Account record was locked. |

## New Account

### Inserting a new account

A call to _BT4QSubmitTask will attempt to insert a new account record when the following parameters are passed:

$1= Application, user, and method name in a text string.

$2= “ACCOUNT_NEW”

$3= Full account data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Account_New_Check”.

#### Successful Preprocessing

If the submitted information passes these tests, then the server assigns the following values to the vBTErrorCode string:

vBTErrorCode =

ID of account(>0) + Tab +“TaskID=” + <Task ID> +Tab + current date +Tab + current time

These values are concatenated with tab characters in between. The creation date and time correspond to when the system was able to start the batch task process.

No matter how long the system is delayed in creating the record, if it eventually creates the record as a result of the information submitted, then this record will be assigned the creation date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this creation date and time to any copy of the account record that is kept in the 4D Open client’s data file. This will enable you to determine if there have been any changes to the created record since this time by examining the date and time of last modifications made to all account records.

#### Unsuccessful Preprocessing

Errors detected when the account is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the account data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the account.

**TABLE 7.** Insert Account Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Account data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new account ID could not be obtained. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |

## Batch Task Processing

The call to insert a account is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record when it returns a positive account ID value. Rather, it has confirmed the account data, obtained the ID value that it will need, and created a Batch Task record.

The call to insert a new account confirms the account data and obtains the ID values that it will need. It then saves the account data in a Batch Task record. The actual table update will be attempted immediately after the ID value is returned to the 4D Open client.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 6. If the data does not pass, as can be determined ahead of time by using the “Account_New_Check” call, then an error is returned and no batch task record is created.

To determine the state of the batch process you canmonitor the vBTErrorText variable, as described in “Brief Overview of the Batch Task System” on page 3.

Even though the supplied data is accepted, the system may not be able to enter the account record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new account for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to a maximum number of attempts that is set by the administrator in 4Q.

In this case the vBTErrorText variable will be assigned values begining with "Batch_Task/n/…" where "n" is the number of the attempt to process the request.

If the system cannot successfully process the account request after the maximum number of attempts has been made, then the account Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this preset age it is marked with the value supplied by the next task submitted. Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. When a task record is overwritten its previous values are lost and it is assigned a new task ID value.

# Modify Account

**modifying an existing account**

A call to _BT4QSubmitTask will attempt to modify an existing account's record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= "ACCOUNT_MODIFY"
$3= Full account data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to "Account_Modify_Check". In fact, the same code that is run when the account data is check is run again before an attempt is made to save the record.

The data submitted to describe the modified account must include all account information, not only that information that you want to modify. If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values.

In order to leave a field unchanged in a current account record you must pass an asterisk, the "*" value, in the corresponding field. Recall that all field values are passed as text regardless of the type of data that is actually saved in the record's field.

The only difference in the case of modifying a account versus adding a account is that the first element in the account data array must contain the ID of the account to be modified. This must be a positive number passed as a string value in the text array that is contained within the Account data blob.

### Successful Preprocessing

When the submitted information passes the data validation tests the server assigns the following values to the vBTErrorCode string:

ID of account + tab + "TaskID=" + tab + modification date + tab + modification time

These values are concatenated with tab characters between them. The modification date and time are the times that will be assigned to the record as such time as the system is able to modify the record.

No matter how long the system is delayed in modifying the record, if it eventually modifies the record as a result of the information submitted, then this

record will be assigned the modification date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this modification date and time to any copy of the account record that is kept in the 4D Open client's data file. This will enable you to determine if there have been any changes to the created record since this time by examining the date and time of last modifications made throughout the account file.

### Unsuccessful Preprocessing

Errors detected when the account is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the account data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the account.

**TABLE 8.** Modify Account Error Codes

| Error Code Str | Description |
|---|---|
| "" (blank) | Submission has not yet completed. |
| value <0 | Account data is unacceptable. |
| "-1" | Unrecognized action parameter. |
| "-2" | No Account exists to match the supplied account ID. |
| "-3" | The passed account ID was not recognized. Check its value and format. |
| "-4" | The account record is locked and cannot be modified. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| "-9" | The submission has been canceled, the Batch Task record has been deleted. |

### Post Processing: the Batch Task

The call to modify an account is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the record when it returns a positive account ID value. Rather, it confirms the account data and creates a Batch Task record. The actual table update will be attempted after the ID value is returned to the 4D Open client.

To determine the state of the batch process you canmonitor the vBTErrorText variable, as described in “Brief Overview of the Batch Task System” on page 3.

Even though the supplied data is accepted, the system may not be able to update the account record. This can happen if the account record is locked by another user.

If the system is not able to update the account for reasons such as this, then the system will make another attempt to update the data after a brief pause. The system will continue trying to update the information up to a hard-coded maximum number of attempts.

If the system cannot successfully process the account insertion request after the maximum number of attempts has been made, then the update account Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. The record will not be lost until a new task reuses the record. When the batch task record is overwritten it is assigned a new ID value.

## Delete Account

### Deleting an account

To delete a account from the 4Q data file call _BT4QSubmitTask with the action code “ACCOUNT_DELETE”. The BLOB must contain a text array that has at least one element. The first element of this array must contain the ID of the account to be erased. No other array values are referenced.

The request to delete a record is handled by the batch task system in the same manner as other batch task requests.

The following error codes are returned in the vBTErrorCode variable.

**TABLE 9.** Delete Account Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Account data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | No Account exists to match the supplied account ID. |
| “-3” | The passed account ID was not recognized. Check its value and format. |
| “-4” | The account record is locked and cannot be deleted. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| Acc’t ID, Task ID, Date, Time | Indicates deletion succeeded. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 4

# Transaction Entry Methods

Update tasks supported by the 4Q API's affecting the "Transaction" table.

4th Quarter supports the following transaction-related tasks through the _BT4QSubmitTask method. Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method.

Each task passes a different "action" parameter and, according to the task, BLOB of data.

**TABLE 10.** Transaction Table Operations

| Action Parameter | Description |
|---|---|
| Transaction_Description | Returning a description of the transaction data format. |
| Transaction_New_Check | Checking the data supplied for a new transaction. |
| Transaction_Modify_Check | Checking the data supplied for a modified transaction. |
| Transaction_New | Inserting a new transaction. |
| Transaction_Modify | Modifying an existing transaction. |
| Transaction_Delete | Deleting a transaction. |

Each task will different information back from the server. Details are described below.

## Transaction Data Format

### Returning a description of the transaction data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new transaction. This information is passed back in the text of the vBTErrorText variable.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “TRANSACTION_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $Method; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
      is opened.
`Note: pass only the names of process variables, not local variables,
            when providing variable references in 4D Open calls.
vIdentity:=Application File+ ";" + Current User +
                                    <name of current method>
vAction:= "Transaction_Description"
$Method:="_BT4QSubmitTask"
$ProcessName:="TestTransaction"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
$4QIsLocked:=False
v4QLockoutFlag:=""
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                  $MyProcessID; "<>vSYDB_LockoutFlag";
                  "v4QLockoutFlag")

If($4DOpenErrCode = 0)
      $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
                  “v4QLockoutFlag”; vSemaphoreState)
      $4QIsLocked:=vSemaphoreState
      $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
```

```
                    v4QLockoutFlag)
End if

Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
      $Method; $Stack128K; $ProcessName;
      $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)`wait 30 seconds max.
      $TicksToWait:=20 `delay between 4D Open calls to prevent
                  `flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (
            MyConnectID; $MyProcessID;"vBTErrorText";
            "vMessage")

      While ((vBTErrorCode = "") &
                        (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID; "vBTErrorCode";
                  "vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorCode";
                  "vBlank")
End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
      End if
```

```
        Case of
        :($BTErrorNumber<0)
              ` Server has returned an error.
              ` Take appropriate response.
        :(vBTErrorCode="")
              ` Timed-out waiting for a response.
              ` Process error here.
        Else
              $TransactionDataSequence:=vMessage
              ` Task completed successfully.
              ` Display transaction data text.
        End case
  End Case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and time are not returned.
- Empty string: processing has not begun.
- Transaction ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the transaction data that 4th Quarter expects to receive as elements in multiple text arrays when a new transaction is created through a 4D Open call. The same information is also available in the vy1BTText and vy2BTText text arrays.

The structure of the vBTErrorText message is as follows:

**TABLE 11.** Structure of the Transaction Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required transaction header text array. |
| 2 | The number of elements in each of the required transaction component text arrays. |
| 3 | A header identifying the following information as pertaining to transaction header data. |
| 4 | A header identifying the following information as pertaining to transaction component data. |
| Starting at 5 | One line descriptions of each of the elements that the transaction header text array, passed in the BLOB variable, is expected to contain. |
| N (depends on version) | At some number of lines down in the text there appears the string "_____ component _____" as an marker that the following text pertains to the structure of the transaction component text array. |
| Starting at N+1 | One line descriptions of each of the elements that the transaction component array, passed in the BLOB variable, is expected to contain. |
| These arrays are stuffed into the blob vBT1Blob. | |

The characters in vBTErrorText before the first carriage return give the size of the transaction header array that is expected. The characters in vBTErrorText before the second carriage return give the size of the transaction component array that is expected.

These values can be extracted by finding the position of the first or second carriage return, extracting the characters lying before them, and converting them to numbers.

Two heading lines follow. After these are one-line descriptions of the contents of each of the transaction header and component array elements. This description is placed in carriage return delimited format. That is, each element of the array is described using a separate line in the vBTErrorText variable.

This same information that is assigned to vBTErrorText is also placed in the text arrays vy1BTText and vy2BTText, which is then stuffed into the blob vBT1Blob. The blob can be read and the arrays extracted from it on the 4D Open client side.

The vy1BTText array contains the text description of the transaction header information. The vy2BTText array contains the text description of the component information. The size of these text arrays can tested using the Size of Array command.

All of the transaction data will need to be placed in text arrays before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text arrays are then packed into a BLOB. This BLOB is passed as $3 when the call is made to create the transaction. Refer to the following section “Inserting a new transaction” on page 66 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

```
23
6
23 element transaction header text array
6 element transaction component text array
Transaction ID
Title
Reference Code
Document Number
Date on Document
Attention
Check Number
Effective Date
Due Date
Type of Entry
Verified (-1=not,0=verified,1=reconciled)
Terms
Allocation key (1=all,2=none,3=per account)
Address
Memo
Internal Value
System Action
Special Handling
Form Code
Order Components By
Related File Number
Related Record ID
Access key (1=general,-1=restricted)
__________ component _________
Account ID
Debit Amount
Payment Priority (1st char must be integer>0)
```

Unit price
Quantity
Component Memo

Execute the “Transaction_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

## Checking Data

### Checking the data supplied with a new transaction

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new or modified transaction record described in the BLOB satisfies 4Q's transaction entry requirements.

This checking requires:

- the transaction information be properly formatted in the BLOB, and
- the data supplied is consistent with 4Q's transaction entry requirements, as are described below.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- it *does not* determine whether or not the transaction record or records related to the transaction, such as accounts, can be immediately created or modified.
- it *does not* lock any records in anticipation for the actual submission of new transaction information.

Getting confirmation from this task does not guarantee that the transaction can be created or modified immediately.

Other concurrent processes may have access to needed counters or accounts when the transaction data is eventually submitted.

If such record locks occur when the transaction is submitted the server will cache the transaction data until updating can be done. This is described further in the following section.

The checking process tests the data for entry as a new or modified transaction. You must pass the correct action parameter along with the data packed in the parameter BLOB.

To start this task to test data for the creation of a new transaction call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "TRANSACTION_NEW_CHECK" (character's case is unimportant)
$3= Full transaction data packed in the BLOB (formatted as described below).

To start this task to test data for the modification of an existing transaction call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= “TRANSACTION_MODIFY_CHECK”
$3= Full transaction data packed in the BLOB (formatted as described below).

The data passed in the BLOB must first be placed in a text array of the correct size. The transaction data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. This code represents a transaction with 2 components. This example implies, but does not explicitly show, the assignment of all the intermediate array elements

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyTransactionHeaderArray;49)
$MyTransactionHeaderArray{1}:="" `ID is supplied by the system.
  ...
$MyTransactionHeaderArray{23}:="1" `sample allocation code.
VARIABLE TO BLOB($MyTransactionHeaderArray;$DataBlob; *)

ARRAY TEXT($MyComponentArray;10)
MyComponentArray{1}:="" `component ID supplied by the system.
...
MyComponentArray{6}:="Payment for PO#9078" `sample comment
VARIABLE TO BLOB( $MyTransactionHeaderArray;$DataBlob;*)
MyComponentArray{1}:=""
...
MyComponentArray{6}:=""
VARIABLE TO BLOB($MyTransactionHeaderArray;$DataBlob; *)
```

The current transaction data structure is given in the following table. The table also provides notes regarding the use or formatting of the data that is supplied. If you are providing data for the modification of a transaction, then you can supply a “*” character (asterisk) to indicate that the previously stored value should be left unchanged. Asterisks are not allowed, however, when specifying the transaction component values.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases where additional conditions apply. These are described in the “Description” column.

Not all fields in the transaction table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

A complete description of the transaction fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the transaction data structure expected in the current version of 4th Quarter that you are communicating with you should execute the "Transaction_Description" call.

**TABLE 12.** Transaction Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| *Transaction Header array elements* | | | |
| 1 | X[0] | Transaction ID | [0]Not used for new transactions. Provide actual > 0 value if referencing or test for modification to an existing transaction. |
| 2 | | Title | |
| 3 | | Reference Code | |
| 4 | | Document Number | |
| 5 | | Date on Document | |
| 6 | | Attention | |
| 7 | | Check Number | |
| 8 | X | Effective Date | must be a valid date. An "*" can be supplied when modifying. |
| 9 | | Due Date | |
| 10 | | Type of Entry | Accepts the value: "" (blank), "PC" period closing, "RE" reversing entry, "RV" foreign currency revaluation entry. |
| 11 | | Verification code (-1,0,1) | for reconciliation and tracking only |
| 12 | | Terms | Must be in 4Q terms format: "Type nnn ppp ddd" where Type should be from 4Q's list "TypesOfTerms", nnn is # days net, ppp is sales % discount, ddd is days for sales discount. |
| 13 | | Allocation key (1,2,3) | Used for transactions affecting AR or AP accounts. 1= to all accounts, 2= to no accounts, 3= as specified on individual accounts. Default of 3 is used if none supplied. |
| 14 | | Address | Address printed on check. |

**TABLE 12.** Transaction Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 15 | | Memo | |
| 16 | | Internal Value | For internal use only. Pass a blank value. |
| 17 | | System Action | Default shipping info is used if none is supplied. |
| 18 | | Special Handling | "NP" of transaction is not to post. Otherwise leave blank. |
| 19 | | Form Code (4 characters). If blank the values "CASH" or "GNRL" are used by default depending on whether the transaction does or does not contain a cash component. | Identifies the form that 4Q can use to open the transaction for modification. "GNRL" for the general entry screen, "CASH" for the cash entry screen. See 4Q Developer Note#15: "Transaction Types" for more information. |
| 20 | | Order Components By | Ascending display order when printed or displayed: "Name" for account name, "Act#", "Item" for component memo, "Dr", "Cr", "TAct" for displaying Debits in account number order, followed by credits. |
| 21 | | Related File Number | Table in the 4Q data file to which the transaction is related. Optional.<br>Assigning a value of -1 or 0 signals 4Q that the user should be able to delete this transaction from the Core accounting areas. Values other than these will prevent deletion from these areas. |
| 22 | | Related Record ID | ID of the record in the 4Q table to which the transaction is related. Optional. |
| 23 | | Access Key (1,-1) | 1=unrestricted, -1=restricted. If not supplied then value is set as restricted whenever any of the listed component accounts are restricted. |
| *Transaction Components Array Elements* | | | |
| 1 | X[2] | Account ID | [2]Must be valid >0 account ID. An "*" can be supplied when modifying. |
| 2 | | Debit Amount | Must be nonzero. Use value >0 for debits, <0 for credits. |
| 3 | | Payment Priority | Only the first character is used to prioritize payment for AR and AP components. Otherwise blank value is used, equivalent to "0", meaning lowest priority. |

**TABLE 12.** Transaction Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 4 | | Unit Price | When supplied, the unit value*quantity should equal the Dr or Cr amount. |
| 5 | | Quantity | |
| 6 | | Component Memo | 80 character text that can be assigned to each component. |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.

**TABLE 13.** Transaction Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -50 | Transaction Header array has <…> elements when it should have <…>. |
| -75 | Transaction component array <…> contains <…> rather than <…> columns. |
| -85 | User's 4Q license has expired. Changes cannot be entered. |
| -110* | Transaction ID <= 0 |
| -115* | Transaction ID= <…> not found. |
| -118* | Unable to locate Transaction with ID=<…> |
| -120* | Transaction ID=<…> has been posted. |
| -125* | Transaction ID=<…> is a check that has been been printed. |
| -160 | No effective transaction date. |
| -165 | Negative check number. |
| -170 | Transaction type does not have 4 characters. |

**TABLE 13.** Transaction Data Value Error

| Error Code | Description |
|---|---|
| -175 | Verified key out of range {1,0,-1} |
| -180 | Allocation key out of range {0,1,2,3} |
| -185 | Access key out of range {1,0,-1}. |
| -190 | Terms of sale =<…> not given in format: "Type DaysDue Discount '/' DaysDiscount |
| -201 | Unspecified problem handling Terms of sale =<…> |
| -210 | Transaction ID could not be obtained or could not be assigned. |
| -230 | Transaction could not be deleted. |
| -235 | Transaction record is locked. |
| -305 | No account specified for component <…> |
| -310 | Component <…> refers to nonexistent account with ID= <…> |
| -320 | Transaction is Unbalanced. Unbalanced transactions are not allowed. |
| -330 | Transaction involves pass through and non-pass through accounts. |

The codes marked with a * only occur when modifying, testing the modification or, or deleting an existing transactions. These values are not supplied to the API when new transactions are created.

## New Transaction

### Inserting a new transaction

A call to _BT4QSubmitTask will attempt to insert a new transaction record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “TRANSACTION_NEW”
$3= Full transaction data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Transaction_New_Check”. It is often preferable to check the transaction data before submitting it as an API call. This will give the caller more control over exception processing.

Errors detected when the transaction is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the transaction data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the transaction.

**TABLE 14.** Insert Transaction Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new transaction ID could not be obtained. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Transaction data is unacceptable. |
| value >0 | The new ID value to be assigned to the transaction. |

### Batch Task Processing

The call to insert an transaction is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record before it returns a positive transaction ID value. Rather, it

confirms the transaction data, obtains the ID values that it will need, and creates a Batch Task record.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 13. If the data does not pass, as can be determined ahead of time by using the “Transaction_New_Check” call, then an error is returned and no batch task record is created.

The error code indicating the request is being processed is of the form:

```
vBTErrorCode =
        Transaction ID(>0) + Tab +“TaskID=” +
        <Task ID> +Tab + current date +Tab + current time
```

The actual insertion of new data into the transaction and related tables begins after this information is assigned to the vBTErrorCode variable.

Even though the supplied data is accepted, the system may not be able to enter the transaction record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new transaction for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to a maximum number of attempts that is set by the administrator in 4Q.

If the system cannot successfully process the transaction request after the maximum number of attempts has been reached, then the transaction Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

Old Batch Task data remain in the Batch Task table until a new task reuses the record.

When overwritten batch task records are given a new Task ID value.

# Modify Transaction

## Modifying an existing transaction

A call to _BT4QSubmitTask will attempt to modify an existing transaction record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “TRANSACTION_MODIFY”
$3= Full transaction data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Transaction_Modify_Check”. It is often preferable to check the transaction data before submitting it. The caller will have more control over exception processing in this case.

The data submitted to describe the modified transaction must include all transaction information. If there are certain fields of the transaction that you do not want to modify then you can pass the asterisk as a text array element. This only applies to values that are not required for locating the transaction and for data that appears in the transaction body. It does not apply to any of the transaction components values.

All component values for a modified transaction must be fully specified. The components of the existing transaction are completely removed. Their accounting effects are reversed. The components of the modified transaction are completely replaced with the line items specified in the API call.

The most obvious difference between the data supplied when modifying a transaction versus adding a transaction is that the first element in the transaction header data array must contain the ID of the transaction to be modified. This must be a positive number passed as a string value in the text array that is contained within the invoice data BLOB.

If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values. Fields in the transaction header that are to be left unchanged must be assigned the asterisk character (“*”).

Errors detected when the transaction is submitted may be reported differently from when the data is submitted for testing. This is because updating a record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the transaction data is not acceptable, the following error code values may be returned if the system is not able to process the request to update the transaction.

**TABLE 15.** Insert Transaction Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-4” | The transaction with the specified ID could not be located. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Transaction data is unacceptable. |
| value >0 | The new ID value to be assigned to the transaction. |

The call to update an transaction is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the existing record before it returns a positive transaction ID value. Rather, it confirms the transaction data and created a Batch Task record The actual table update begins just after the confirming transaction ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to update the transaction record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new transaction for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to a hard-coded maximum number of attempts.

If the system cannot successfully process the transaction update request after the maximum number of attempts has been reached, then the update transaction Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and

- the result of the systems attempt to process them.

The administrator can then

- print
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the 4D Open page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

The contents of exiisting records are not lost until they are reused. Task ID values are reassigned as well at this time.

## Delete Transaction

### Deleting a transaction

To delete an transaction from the 4Q data file call _BT4QSubmitTask with the action code “TRANSACTION_DELETE”. The BLOB must conatin a text array that has at least one element. The first element of this array must contain the ID of the ID of the transaction to be errased. No other array values are referenced.

The request to delete a record is handled by the batch task system in the same manner as other batch task requests.

The following error codes are returned in the vBTErrorCode variable.

**TABLE 16.** Delete Transaction Error Codes

| Error Code Str | Description |
|---|---|
| ““ (blank) | Submission has not yet completed. |
| value <0 | Transaction deletion error, refer to message in vBTErrorText. |
| “-1” | Unrecognized action parameter. |
| “-2” | No transaction exists to match the supplied transaction ID. |
| “-3” | The transaction ID cannot be recognized. Check its value and format. |
| “-4” | The transaction is locked and cannot be deleted. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| Trans ID, Task ID, Date, Time | Indicates deletion succeeded. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 5

# Customer Data Entry Methods

Update tasks supported by the 4Q API's affecting the "Customer" table.

4th Quarter supports customer-related tasks through the _BT4QSubmitTask method.

Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method. Each task passes a different "action" parameter and, according to the task, a different BLOB of data.

**TABLE 17.** Customer Table Operations

| Action Parameter | Description |
|---|---|
| Customer_Description | Returning a description of the customer data format. |
| Customer_New_Check | Checking the data supplied for creating a new customer. |
| Customer_Modify_Check | Checking the data supplied for modifying an existing customer. |
| Customer_New | Inserting a new customer. |
| Customer_Modify | Modifying an existing customer. |
| Customer_Delete | Deleting a customer. |

Each task will different information back from the server. Details are described below.

## Customer Data Format

### Returning a description of the customer data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new customer. This information is passed back in the text of the vBTErrorText variable.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “CUSTOMER_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $Method; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
                                    is opened.
`Note: pass only the names of process variables, not local variables,
            when providing variable references in 4D Open calls.
vIdentity:=Application File+ ";" + Current User +
                                    <name of current method>
vAction:= "Customer_Description"
$Method:="_BT4QSubmitTask"
$ProcessName:="TestCustomer"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
v4QLockoutFlag:=""
$4QIsLocked:=False
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                    $MyProcessID; "<>vSYDB_LockoutFlag";
                    "v4QLockoutFlag")

If($4DOpenErrCode = 0)
      $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
                    "v4QLockoutFlag"; vSemaphoreState)
      $4QIsLocked:=vSemaphoreState
      $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
```

```
                    "v4QLockoutFlag")
End if

Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
            $Method; $Stack128K;$ProcessName;
            $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)
                              `wait 30 seconds max.
      $TicksToWait:=20        `delay between 4D Open calls to
                              `prevent flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                  $MyProcessID;"vBTErrorText"; "vMessage")

      While ((vBTErrorCode = "") &
                              (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID; "vBTErrorCode";
                  "vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID; "vBTErrorCode";
                  "vBlank")
      End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
      End if
```

```
Case of
:($BTErrorNumber<0)
      ` Server has returned an error.
      ` Take appropriate response.
:(vBTErrorCode="")
      ` Timed-out waiting for a response.
      ` Process error here.
Else
      $CustomerDataSequence:=vMessage
      ` Task completed successfully.
      ` Display customer data text.
End case
End Case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and time are not returned.
- Empty string: processing has not begun.
- Customer ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the customer data that 4th Quarter expects to receive as elements in a text array when you submit a new customer through a 4D Open call. The same information is also available in the vy1BTText array which has been stuffed into the vBT1Blob variable.

The structure of the vBTErrorText message is as follows:

**TABLE 18.** Structure of the Customer Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required text array. |
| 2 | A header identifying the following information as pertaining to customer data. |
| Starting at 3 | One line descriptions of each of the elements that the customer text array, passed in the BLOB variable, is expected to contain. |
| This array is stuffed in the blob vBT1Blob. | |

The characters in vBTErrorText before the first carriage return give the size of the array that is expected. This value can be extracted by finding the position of the first carriage return, extracting the characters lying before it, and converting them to a number.

A heading line follows. After that there are one-line descriptions of each of the customer array elements. These descriptions are placed in carriage return delimited format. That is, each element of the customer array is described using a separate line in the vBTErrorText variable.

This same information that is assigned to vBTErrorText is also placed in the text array vy1BTText. The size of this text array can tested using the Size of Array command. which is then stuffed into the blob vBT1Blob. The blob can be read and the array extracted from it on the 4D Open client side.

All of the customer data will need to be placed in a text array before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text array must then be packed into a BLOB. This BLOB is passed as $3 when the call is made to create the customer. Refer to the following section “Inserting a new customer” on page 86 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

```
64
64 element customer text array
Customer ID
```

Customer Code
Customer Type
Customer Status
Name of user who created
Salutation
First Name
Last Name
Title
Company
Department
Credit Card Number
Credit Card Type
Credit Card Expire Date
Discount Percent
Tax Exemption Number
Concatenated Tax ID's, each followed by '/'
Terms of Sale
Finance Percent Per Month
Price Code, for price lookup
Ship Via Preference
Phone, work
Fax, work
Phone, home
Fax, home
Memo
Web Email
Web Password
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
Address A, billing
Address B, billing
Address C, billing
City, billing
State, billing
Postal Code, billing

Zip Plus 4, billing
Country, billing
URL, billing
Email, billing
<unused>
<unused>
<unused>
<unused>
<unused>
Address A, shipping
Address B, shipping
Address C, shipping
City, shipping
State, shipping
Postal Code, shipping
Zip Plus 4, shipping
Country, shipping
URL, shipping
Email, shipping

Execute the “Customer_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

## Checking Data

### Checking the data supplied for a new customer

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new customer record described in the BLOB satisfies 4Q's customer entry requirements.

This checking requires:

- the customer information be properly formatted in the BLOB, and
- the data supplied is consistent with 4Q's customer entry requirements, as are described below.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- it *does not* determine whether or not the customer record or records related to the customer, such as accounts and addresses, can be immediately created.
- it *does not* lock any records in anticipation for the actual submission of new customer information.

Getting confirmation from this task does not guarantee that the customer can be modified or created immediately.

Other concurrent processes may have access to needed accounts or counters at the time when the customer data is eventually submitted.

If such record locks occur when the customer is submitted the server will cache the customer data until updating can be done. This is described further in the following section.

To start this task to create a new customer call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "CUSTOMER_NEW_CHECK"
$3= Full customer data packed in the BLOB (formatted as described below).

To start this task to modify and existing customer call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "CUSTOMER_MODIFY_CHECK"
$3= Full customer data packed in the BLOB (formatted as described below).

The data passed in the BLOB must first be placed in a text array of the correct size. The customer data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. In this example implies, but does not explicitly show, the assignment of all the intermediate array elements.

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyCustomerArray;46)
$MyCustomerArray{1}:="0"
  ...
$MyCustomerArray{46}:="TheAddress@TheDomain.com"
VARIABLE TO BLOB($MyCustomerArray; $DataBlob)
```

The current customer data structure is given in table Table 19. The table also provides notes regarding the use or formatting of the data that must be supplied.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases one item or another item is required. This is described further in the “Description” column.

Not all fields in the customer table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

A complete description of the customer fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the customer data structure expected in the current version of 4th Quarter that you are communicating with you should execute the “Customer_Description” call.

**TABLE 19.** Customer Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 1 | X[0] | Customer ID | [0]Not used for a new customer. Must be the actual > 0 value for an existing customer. |
| 2 | X | Customer Code | User defined. |

**TABLE 19.** Customer Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 3 | X | Customer Type | Item from 4Q's "CustomerTypes" list. |
| 4 | | Customer Status | Item from 4Q's "CustomerStatus" list. |
| 5 | | Name of user creating record | |
| 6 | | Salutation | |
| 7 | | First Name | |
| 8 | X[1] | Last Name | [1]Must have either a last or company name. |
| 9 | | Title | |
| 10 | X[1] | Company | [1]Must have either a company or last name. |
| 11 | | Department | |
| 12 | | Credit Card Number | Containing numeric characters only. |
| 13 | | Credit Card Type | Any value accepted, but it should be an item from 4Q's "CreditCards" list to preserve consistency. |
| 14 | | Credit Card Expire Date | Must be in format ##/####. |
| 15 | | Discount Percent | |
| 16 | | Tax Exemption Number | |
| 17 | | Concatenated Tax ID's | Each ID must be followed by '/' |
| 18 | | Terms of Sale | In format "Type nnn ppp ddd" where Type should be from 4Q's list "TypesOfTerms", nnn is # days net, ppp is sales % discount, ddd is days for sales discount. |
| 19 | | Finance Percent Per Month | |
| 20 | | Price Code, for price lookup | Used only if it matches a price code carried by an inventory item. |
| 21 | | Ship Via Preference | Any value accepted, but it should be an item from 4Q's "ShippingCarriers" list. |
| 22 | | Phone, work | With formatting embedded. |
| 23 | | Fax, work | With formatting embedded. |
| 24 | | Phone, home | With formatting embedded. |

**TABLE 19.** Customer Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 25 | | Fax, home | With formatting embedded. |
| 26 | | Memo | |
| 27 | | Web Email | |
| 28 | | Web Password | |
| | | 29 – 39 unused | |
| 40 | X[2] | Address A, billing | [2]Must have at least one Address line. |
| 41 | X[2] | Address B, billing | |
| 42 | X[2] | Address C, billing | |
| 43 | X[3] | City, billing | [3]Must have city or state. |
| 44 | X[3] | State, billing | |
| 45 | | Postal Code, billing | |
| 46 | | Zip Plus 4, billing | |
| 47 | | Country, billing | |
| 48 | | URL, billing | |
| 49 | | Email, billing | |
| | | 50 – 54 unused | |
| 55 | X[4] | Address A, shipping | [4]Must have at least one Address line. |
| 56 | X[4] | Address B, shipping | |
| 57 | X[4] | Address C, shipping | |
| 58 | X[5] | City, shipping | [5]Must have city or state. |
| 59 | X[5] | State, shipping | |
| 60 | | Postal Code, shipping | |
| 61 | | Zip Plus 4, shipping | |
| 62 | | Country, shipping | |

**TABLE 19.** Customer Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 63 | | URL, shipping | |
| 64 | | Email, shipping | |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.

**TABLE 20.** Customer Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -115 | Bad customer ID <ID=…> (Only applies when modifying) |
| -118 | The customer with ID= <…> could not be found. (Only applies when modifying) |
| -120 | No customer code. |
| -125 | The customer code contains the suffix <…> that is reserved for system use. |
| -130 | Duplicate customer code: <…> |
| -132 | The customer type is blank. |
| -133 | The customer type <…> does not support a receivable account. |
| -135 | The customer has neither last name or company name. At least one must be provided. |
| -140 | Customer ID= <…> has no billing street address. |
| -145 | Customer ID= <…> missing billing city and state. |
| -150 | Customer ID= <…> has no shipping street address. |
| -155 | Customer ID= <…> missing shipping city and state. |

**TABLE 20.** Customer Data Value Error

| Error Code | Description |
|---|---|
| -156 | Customer ID= <…> Bad credit card expiration date or date format. |
| -160 | Sales tax ID <=0. |
| -165 | Sales tax ID= <…> cannot be found. |
| -170 | Sales tax ID= <…> has no related account. |
| -175 | Account ID = <…> " for Sales tax ID= <…> cannot be found. |
| -180 | Terms of sale = <…> not given in format: Type DaysDue Discount '/' DaysDiscount |
| -210 | Customer ID could not be obtained or could not be assigned. |
| -215 | Necessary account ID could not be obtained or could not be assigned. |
| -230 | Customer record with ID = <…> cannot be deleted. |
| -235 | Customer record with ID = <…> is locked. |

## New Customer

### Inserting a new customer

A call to _BT4QSubmitTask will attempt to insert a new customer record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “CUSTOMER_NEW”
$3= Full customer data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Customer_New_Check”. In fact, the same code that is run when the customer data is check is run again before an attempt is made to save the record.

#### Successful Preprocessing

If the submitted information passes these tests, then the server assigns the following values to the vBTErrorCode string:

Customer ID + tab + “TaskID=” + task ID + tab + creation date + tab + creation time

The values are concatenated with tab characters as separators. The creation date and time are the times that will be assigned to the record as such time as the system is able to create the record.

No matter how long the system is delayed in creating the record, if it eventually creates the record as a result of the information submitted, then this record will be assigned the creation date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this creation date and time to any copy of the customer record that is kept in the 4D Open client’s data file. This will enable you to determine if there have been any changes to the created record since this time by examining the date and time of last modifications made to all customer records.

It is often preferable to be able to check the customer data before submitting it as the 4D Open will have more control over exception processing.

#### Unsuccessful Preprocessing

Errors detected when the customer is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new

record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the customer data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the customer.

**TABLE 21.** Insert Customer Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Customer data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new customer ID could not be obtained. |
| “-3” | A new customer receivable account ID could not be obtained. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| ID, Task ID, Date, Time | Indicates task has been accepted and is being, or has been processed. |

## Batch Task Processing

The call to insert or modify a customer is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record when it returns a positive customer ID value. Rather, it has confirmed the customer data, obtained the ID value that it will need, and created a Batch Task record.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 20. If the data does not pass, as can be determined ahead of time by using the “Customer_New_Check” call, then an error is returned and no batch task record is created.

The call to insert a new customer confirms the customer data and obtains the ID values that it will need. It then saves the customer data in a Batch Task record. The actual table update is attempted after the ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to enter the customer record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to create or modify the customer for reasons such as this, then the system will make another attempt to enter the data after a brief pause whose duration is set by the 4Q administrator. The system will continue trying to save the information up to a maximum number of attempts that is set by the administrator in 4Q.

If the system cannot successfully process the customer request after the maximum number of attempts has been made, then the customer Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this preset age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

However, the record will not be deleted. It will remain in the Batch Task table, marked for reused, until a new task reuses the record.

Reused batch task records are assigned new task ID's.

## *Modify Customer*

**modifying an existing customer**

A call to _BT4QSubmitTask will attempt to modify an existing customer's record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= "CUSTOMER_MODIFY"
$3= Full customer data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to "Customer_Modify_Check". In fact, the same code that is run when the customer data is check is run again before an attempt is made to save the record.

The data submitted to describe the modified customer must include all customer information, not only that information that you want to modify. If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values.

The only difference in the case of modifying a customer versus adding a customer is that the first element in the customer data array must contain the ID of the customer to be modified. This must be a positive number passed as a string value in the text array that is contained within the Customer data blob.

### Successful Preprocessing

When the submitted information passes the data validation tests the server assigns the following values to the vBTErrorCode string:

ID of customer + tab + "TaskID=" + task ID + tab + modification date + tab + modification time

These values are concatenated with tab characters between them. The modification date and time are the times that will be assigned to the record as such time as the system is able to modify the record.

No matter how long the system is delayed in modifying the record, if it eventually modifies the record as a result of the information submitted, then this record will be assigned the modification date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this modification date and time to any copy of the customer record that is kept in the 4D Open client's data file. This will enable you to determine if there have been any changes to the created record since this

time by examining the date and time of last modifications made throughout the customer file.

### Unsuccessful Preprocessing

Errors detected when the customer is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the customer data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the customer.

**TABLE 22.** Insert Customer Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Customer data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-4” | Customer with specified ID could not be located for modification. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| ID, Task ID, Date, Time | The task has been accepted and is either being or has been processed. |

### Post Processing: the Batch Task

The call to modify a customer is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the record when it returns a positive customer ID value. Rather, it has confirmed the customer data and created a Batch Task record.

The call to update an existing customer first confirms the full customer data provided. It then saves the customer data in a Batch Task record. The actual table update begins after the ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to update the customer record. This can happen if the customer record is locked by another user.

If the system is not able to update the customer for reasons such as this, then the system will make another attempt to update the data after a brief pause. The system will continue trying to update the information up to a maximum number of attempts that is set by the 4Q administrator.

If the system cannot successfully process the customer request after the maximum number of attempts has been made, then the customer Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the 4D Open page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

The batch task data remain in the Batch Task table until a new task reuses the record. Reused records are assigned new task ID's.

## Delete Customer

### Deleting a customer

To delete a customer from the 4Q data file call _BT4QSubmitTask with the action code “CUSTOMER_DELETE”. The BLOB must contain a text array that has at least one element. The first element of this array must contain the ID of the customer to be erased. No other array values are referenced.

The following error codes may be returned in the vBTErrorCode variable.

**TABLE 23.** Delete Customer Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Customer data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | No customer exists to match the supplied customer ID. |
| “-3” | The passed customer ID was not recognized. Check its value and format. |
| “-4” | The customer record is locked and cannot be deleted. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 6

# Vendor Data Entry Methods

Update tasks supported by the 4Q API's affecting the "Vendor" table.

4th Quarter supports vendor-related tasks through the _BT4QSubmitTask method.

Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method. Each task passes a different "action" parameter and, according to the task, a different BLOB of data.

**TABLE 24.** Vendor Table Operations

| Action Parameter | Description |
|---|---|
| Vendor_Description | Returning a description of the vendor data format. |
| Vendor_New_Check | Checking the data supplied for creating a new vendor. |
| Vendor_Modify_Check | Checking the data supplied for modifying an existing vendor. |
| Vendor_New | Inserting a new vendor. |
| Vendor_Modify | Modifying an existing vendor. |
| Vendor_Delete | Deleting a vendor. |

Each task will different information back from the server. Details are described below.

## Vendor Data Format

### Returning a description of the vendor data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new vendor. This information is passed back in the text of the vBTErrorText variable.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “VENDOR_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $Method; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
                                    is opened.
`Note: pass only the names of process variables, not local variables,
            when providing variable references in 4D Open calls.
vIdentity:=Application File+ ";" + Current User +
                                    <name of current method>
vAction:= "Vendor_Description"
$Method:="_BT4QSubmitTask"
$ProcessName:="TestVendor"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
v4QLockoutFlag:=""
$4QIsLocked:=False
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                    $MyProcessID; "<>vSYDB_LockoutFlag";
                    "v4QLockoutFlag")

If($4DOpenErrCode = 0)
      $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
                    "v4QLockoutFlag"; vSemaphoreState)
      $4QIsLocked:=vSemaphoreState
      $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
```

```
                    "v4QLockoutFlag")
End if

Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
            $Method; $Stack128K;$ProcessName;
            $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)
                              `wait 30 seconds max.
      $TicksToWait:=20        `delay between 4D Open calls to
                              `prevent flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                  $MyProcessID;"vBTErrorText"; "vMessage")

      While ((vBTErrorCode = "") &
                              (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID; "vBTErrorCode";
                  "vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID; "vBTErrorCode";
                  "vBlank")
      End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
      End if
```

```
        Case of
        :($BTErrorNumber<0)
              ` Server has returned an error.
              ` Take appropriate response.
        :(vBTErrorCode="")
              ` Timed-out waiting for a response.
              ` Process error here.
        Else
              $VendorDataSequence:=vMessage
              ` Task completed successfully.
              ` Display vendor data text.
        End case
  End Case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and time are not returned.
- Empty string: processing has not begun.
- Vendor ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the vendor data that 4th Quarter expects to receive as elements in a text array when you submit a new vendor through a 4D Open call. The same information is also available in the vy1BTText array which has been stuffed into the vBT1Blob variable.

The structure of the vBTErrorText message is as follows:

**TABLE 25.** Structure of the Vendor Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required text array. |
| 2 | A header identifying the following information as pertaining to vendor data. |
| Starting at 3 | One line descriptions of each of the elements that the vendor text array, passed in the BLOB variable, is expected to contain. |
| This array is stuffed in the blob vBT1Blob. | |

The characters in vBTErrorText before the first carriage return give the size of the array that is expected. This value can be extracted by finding the position of the first carriage return, extracting the characters lying before it, and converting them to a number.

A heading line follows. After that there are one-line descriptions of each of the vendor array elements. These descriptions are placed in carriage return delimited format. That is, each element of the vendor array is described using a separate line in the vBTErrorText variable.

This same information that is assigned to vBTErrorText is also placed in the text array vy1BTText. The size of this text array can tested using the Size of Array command. which is then stuffed into the blob vBT1Blob. The blob can be read and the array extracted from it on the 4D Open client side.

All of the vendor data will need to be placed in a text array before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text array must then be packed into a BLOB. This BLOB is passed as $3 when the call is made to create the vendor. Refer to the following section “Inserting a new vendor” on page 106 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

```
64
64 element vendor text array
Vendor ID
```

Vendor Code
Vendor Type
Vendor Status
Name of user who created
Salutation
First Name
Last Name
Title
Company
Department
Commission Percent
<unused>
<unused>
Discount Percent
Tax Exemption Number
Concatenated Tax ID's, each followed by '/'
Terms of Sale
Finance Percent Per Month
Price Code, for price lookup
Ship Via Preference
Phone, work
Fax, work
Phone, home
Fax, home
Memo
Web Email
Web Password
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
<unused>
Address A, billing
Address B, billing
Address C, billing
City, billing
State, billing
Postal Code, billing

Zip Plus 4, billing
Country, billing
URL, billing
Email, billing
<unused>
<unused>
<unused>
<unused>
<unused>
Address A, shipping
Address B, shipping
Address C, shipping
City, shipping
State, shipping
Postal Code, shipping
Zip Plus 4, shipping
Country, shipping
URL, shipping
Email, shipping

Execute the “Vendor_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

## Checking Data

### Checking the data supplied for a new vendor

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new vendor record described in the BLOB satisfies 4Q's vendor entry requirements.

This checking requires:

- the vendor information be properly formatted in the BLOB, and
- the data supplied is consistent with 4Q's vendor entry requirements, as are described below.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- it *does not* determine whether or not the vendor record or records related to the vendor, such as accounts and addresses, can be immediately created.
- it *does not* lock any records in anticipation for the actual submission of new vendor information.

Getting confirmation from this task does not guarantee that the vendor can be modified or created immediately.

Other concurrent processes may have access to needed accounts or counters at the time when the vendor data is eventually submitted.

If such record locks occur when the vendor is submitted the server will cache the vendor data until updating can be done. This is described further in the following section.

To start this task to create a new vendor call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "VENDOR_NEW_CHECK"
$3= Full vendor data packed in the BLOB (formatted as described below).

To start this task to modify and existing vendor call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "VENDOR_MODIFY_CHECK"
$3= Full vendor data packed in the BLOB (formatted as described below).

The data passed in the BLOB must first be placed in a text array of the correct size. The vendor data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. In this example implies, but does not explicitly show, the assignment of all the intermediate array elements.

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyVendorArray;46)
$MyVendorArray{1}:="0"
  ...
$MyVendorArray{46}:="TheAddress@TheDomain.com"
VARIABLE TO BLOB($MyVendorArray; $DataBlob)
```

The current vendor data structure is given in table Table 26. The table also provides notes regarding the use or formatting of the data that must be supplied.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases one item or another item is required. This is described further in the “Description” column.

Not all fields in the vendor table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

A complete description of the vendor fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the vendor data structure expected in the current version of 4th Quarter that you are communicating with you should execute the “Vednor_Description” call.

**TABLE 26.** Vendor Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 1 | X[0] | Vendor ID | [0]Not used for a new vendor. Must be the actual > 0 value for an existing vendor. |
| 2 | X | Vendor Code | User defined. |

**TABLE 26.** Vendor Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 3 | X | Vendor Type | Item from 4Q's "VendorTypes" list. |
| 4 | | Vendor Status | Item from 4Q's "VendorStatus" list. |
| 5 | | Name of user creating record | |
| 6 | | Salutation | |
| 7 | | First Name | |
| 8 | X[1] | Last Name | [1]Must have either a last or company name. |
| 9 | | Title | |
| 10 | X[1] | Company | [1]Must have either a company or last name. |
| 11 | | Department | |
| 12 | | Commission Percent | Containing numeric characters only. |
| 13 | | 13– 14 unused | |
| 15 | | Discount Percent | |
| 16 | | Tax Exemption Number | |
| 17 | | Concatenated Tax ID's | Each ID must be followed by '/' |
| 18 | | Terms of Sale | In format "Type nnn ppp ddd" where Type should be from 4Q's list "TypesOfTerms", nnn is # days net, ppp is sales % discount, ddd is days for sales discount. |
| 19 | | Finance Percent Per Month | |
| 20 | | Price Code, for price lookup | Used only if it matches a price code carried by an inventory item. |
| 21 | | Ship Via Preference | Any value accepted, but it should be an item from 4Q's "ShippingCarriers" list. |
| 22 | | Phone, work | With formatting embedded. |
| 23 | | Fax, work | With formatting embedded. |
| 24 | | Phone, home | With formatting embedded. |
| 25 | | Fax, home | With formatting embedded. |
| 26 | | Memo | |

**TABLE 26.** Vendor Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 27 | | Web Email | |
| 28 | | Web Password | |
| | | 29 – 39 unused | |
| 40 | X[2] | Address A, billing | [2]Must have at least one Address line. |
| 41 | X[2] | Address B, billing | |
| 42 | X[2] | Address C, billing | |
| 43 | X[3] | City, billing | [3]Must have city or state. |
| 44 | X[3] | State, billing | |
| 45 | | Postal Code, billing | |
| 46 | | Zip Plus 4, billing | |
| 47 | | Country, billing | |
| 48 | | URL, billing | |
| 49 | | Email, billing | |
| | | 50 – 54 unused | |
| 55 | X[4] | Address A, shipping | [4]Must have at least one Address line. |
| 56 | X[4] | Address B, shipping | |
| 57 | X[4] | Address C, shipping | |
| 58 | X[5] | City, shipping | [5]Must have city or state. |
| 59 | X[5] | State, shipping | |
| 60 | | Postal Code, shipping | |
| 61 | | Zip Plus 4, shipping | |
| 62 | | Country, shipping | |
| 63 | | URL, shipping | |
| 64 | | Email, shipping | |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.

**TABLE 27.** Vendor Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -115 | Bad vendor ID <ID=…> (Only applies when modifying) |
| -118 | The vendor with ID= <…> could not be found. (Only applies when modifying) |
| -120 | No vendor code. |
| -125 | The vendor code contains the suffix <…> that is reserved for system use. |
| -130 | Duplicate vendor code: <…> |
| -132 | The vendor type is blank. |
| -135 | The vendor has neither last name or company name. At least one must be provided. |
| -140 | Vendor ID= <…> has no billing street address. |
| -145 | Vendor ID= <…> missing billing city and state. |
| -150 | Vendor ID= <…> has no shipping street address. |
| -155 | Vendor ID= <…> missing shipping city and state. |
| -156 | Vendor ID= <…> Bad credit card expiration date or date format. |
| -160 | Sales tax ID <=0. |
| -165 | Sales tax ID= <…> cannot be found. |
| -170 | Sales tax ID= <…> has no related account. |
| -175 | Account ID = <…> " for Sales tax ID= <…> cannot be found. |

**TABLE 27.** Vendor Data Value Error

| Error Code | Description |
|---|---|
| -180 | Terms of sale = <…> not given in format: Type DaysDue Discount '/' DaysDiscount |
| -210 | Vendor ID could not be obtained or could not be assigned. |
| -215 | Necessary account ID could not be obtained or could not be assigned. |
| -230 | Vendor record with ID = <…> cannot be deleted. |
| -235 | Vendor record with ID = <…> is locked. |

## New Vendor

### Inserting a new vendor

A call to _BT4QSubmitTask will attempt to insert a new vendor record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “VENDOR_NEW”
$3= Full vendor data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Vendor_New_Check”. In fact, the same code that is run when the vendor data is check is run again before an attempt is made to save the record.

#### Successful Preprocessing

If the submitted information passes these tests, then the server assigns the following values to the vBTErrorCode string:

Vendor ID + tab + “TaskID=” + task ID + tab + creation date + tab + creation time

The values are concatenated with tab characters as separators. The creation date and time are the times that will be assigned to the record as such time as the system is able to create the record.

No matter how long the system is delayed in creating the record, if it eventually creates the record as a result of the information submitted, then this record will be assigned the creation date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this creation date and time to any copy of the vendor record that is kept in the 4D Open client’s data file. This will enable you to determine if there have been any changes to the created record since this time by examining the date and time of last modifications made to all vendor records.

It is often preferable to be able to check the vendor data before submitting it as the 4D Open will have more control over exception processing.

#### Unsuccessful Preprocessing

Errors detected when the vendor is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new

record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the vendor data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the vendor.

**TABLE 28.** Insert Vendor Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Vendor data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new vendor ID could not be obtained. |
| “-3” | A new vendor payable account ID could not be obtained. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| ID, Task ID, Date, Time | Indicates task has been accepted and is being, or has been processed. |

## Batch Task Processing

The call to insert or modify a vendor is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record when it returns a positive vendor ID value. Rather, it has confirmed the vendor data, obtained the ID value that it will need, and created a Batch Task record.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 28. If the data does not pass, as can be determined ahead of time by using the “Vendor_New_Check” call, then an error is returned and no batch task record is created.

The call to insert a new vendor confirms the vendor data and obtains the ID values that it will need. It then saves the vendor data in a Batch Task record. The actual table update is attempted after the ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to enter the vendor record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to create or modify the vendor for reasons such as this, then the system will make another attempt to enter the data after a brief pause whose duration is set by the 4Q administrator. The system will continue trying to save the information up to a maximum number of attempts that is set by the administrator in 4Q.

If the system cannot successfully process the vendor request after the maximum number of attempts has been made, then the vendor Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this preset age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

However, the record will not be deleted. It will remain in the Batch Task table, marked for reused, until a new task reuses the record.

Reused batch task records are assigned new task ID's.

# Modify Vendor

**modifying an existing vendor**

A call to _BT4QSubmitTask will attempt to modify an existing vendor's record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= "VENDOR_MODIFY"
$3= Full vendor data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to "Vendor_Modify_Check". In fact, the same code that is run when the vendor data is check is run again before an attempt is made to save the record.

The data submitted to describe the modified vendor must include all vendor information, not only that information that you want to modify. If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values.

The only difference in the case of modifying a vendor versus adding a vendor is that the first element in the vendor data array must contain the ID of the vendor to be modified. This must be a positive number passed as a string value in the text array that is contained within the Vendor data blob.

### Successful Preprocessing

When the submitted information passes the data validation tests the server assigns the following values to the vBTErrorCode string:

ID of vendor + tab + "TaskID=" + task ID + tab + modification date + tab + modification time

These values are concatenated with tab characters between them. The modification date and time are the times that will be assigned to the record as such time as the system is able to modify the record.

No matter how long the system is delayed in modifying the record, if it eventually modifies the record as a result of the information submitted, then this record will be assigned the modification date and times indicated in the vBTErrorCode variable at the time when the task was first submitted.

You should assign this modification date and time to any copy of the vendor record that is kept in the 4D Open client's data file. This will enable you to determine if there have been any changes to the created record since this

time by examining the date and time of last modifications made throughout the vendor file.

### Unsuccessful Preprocessing

Errors detected when the vendor is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the vendor data is not acceptable, the following error code values may be returned if the system is not able to process the request to insert the vendor.

**TABLE 29.** Insert Vendor Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Vendor data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-4” | Vendor with specified ID could not be located for modification. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| ID, Task ID, Date, Time | The task has been accepted and is either being or has been processed. |

### Post Processing: the Batch Task

The call to modify a vendor is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the record when it returns a positive vendor ID value. Rather, it has confirmed the vendor data and created a Batch Task record.

The call to update an existing vendor first confirms the full vendor data provided. It then saves the vendor data in a Batch Task record. The actual table update begins after the ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to update the vendor record. This can happen if the vendor record is locked by another user.

If the system is not able to update the vendor for reasons such as this, then the system will make another attempt to update the data after a brief pause. The system will continue trying to update the information up to a maximum number of attempts that is set by the 4Q administrator.

If the system cannot successfully process the vendor request after the maximum number of attempts has been made, then the vendor Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the 4D Open page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point.

The batch task data remain in the Batch Task table until a new task reuses the record. Reused records are assigned new task ID's.

## Delete Vendor

### Deleting a vendor

To delete a vendor from the 4Q data file call _BT4QSubmitTask with the action code “VENDOR_DELETE”. The BLOB must contain a text array that has at least one element. The first element of this array must contain the ID of the vendor to be erased. No other array values are referenced.

The following error codes may be returned in the vBTErrorCode variable.

**TABLE 30.** Delete Vendor Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Vendor data is unacceptable. |
| “-1” | Unrecognized action parameter. |
| “-2” | No vendor exists to match the supplied vendor ID. |
| “-3” | The passed vendor ID was not recognized. Check its value and format. |
| “-4” | The vendor record is locked and cannot be deleted. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 7

# Invoice Data Entry Methods

Update tasks supported by the 4Q API's affecting the "vInvoice" table.

4th Quarter supports the following invoice-related tasks through the _BT4QSubmitTask method. Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method.

Each task passes a different "action" parameter and, according to the task, BLOB of data.

**TABLE 31.** Invoice Table Operations

| Action Parameter | Description |
|---|---|
| Invoice_Description | Returning a description of the invoice data format. |
| Invoice_New_Check | Checking the data supplied for a new invoice. |
| Invoice_Modify_Check | Checking the data supplied for the modification of an existing invoice. |
| Invoice_New | Inserting a new invoice. |
| Invoice_Modify | Modifying an existing invoice. |
| Invoice_Delete | Deleting an invoice. |

Each task will different information back from the server. Details are described below.

## Invoice Data Format

### Returning a description of the invoice data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new invoice. This information is passed back in the text of the vBTError-Text variable.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “INVOICE_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $Method; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
                                    is opened.
vIdentity:=Application File+ ";" + Current User +
                                    <name of current method>
vAction:= "Invoice_Description"
$Method:="_BT4QSubmitTask"
$ProcessName:="TestInvoice"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
v4QLockoutFlag:=""
$4QIsLocked:=False
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                    $MyProcessID; "<>vSYDB_LockoutFlag";
                    "v4QLockoutFlag")

If($4DOpenErrCode = 0)
      $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
                    "<>vSYDB_LockoutFlag"; vSemaphoreState)
      $4QIsLocked:=vSemaphoreState
      $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
                    "<>vSYDB_LockoutFlag")
End if
```

```
Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
      $Method; $Stack128K; $ProcessName;
      $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)`wait 30 seconds max.
      $TicksToWait:=20 `delay between 4D Open calls to prevent
                  `flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (MyConnectID;
            $MyProcessID;"vBTErrorText"; "vMessage")

      While ((vBTErrorCode = "") &
                        (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; "$MyProcessID";
                  "vBTErrorCodeCode";"vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; "$MyProcessID";"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorCode";
                  "vBlank")
End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
      End if

      Case of
```

```
        :($BTErrorNumber<0)
              ` Server has returned an error.
              ` Take appropriate response.
        Case of
        :(Num(vBTErrorCode)<0)
              ` Server has returned an error.
              ` Take appropriate response.
        :(vBTErrorCode="")
              ` Timed-out waiting for a response.
              ` Process error here.
        Else
              $InvoiceDataSequence:=vMessage
              ` Task completed successfully.
              ` Display invoice data text.
        End case
  End case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and time are not returned.
- Empty string: processing has not begun.
- Invoice ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the invoice data that 4th Quarter expects to receive as elements in multiple text arrays when a new invoice is created through a 4D Open call.

The structure of the vBTErrorText message is as follows:

**TABLE 32.** Structure of the Invoice Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required invoice header text array. |
| 2 | The number of elements in each of the required invoice line item text arrays. |
| 3 | A header identifying the following information as pertaining to invoice header data. |
| 4 | A header identifying the following information as pertaining to invoice line item data. |
| Starting at 5 | One line descriptions of each of the elements that the invoice header text array, passed in the BLOB variable, is expected to contain. |
| N (depends on version) | At some number of lines down in the text there appears the string "_____ line item _____" as an marker that the following text pertains to the structure of the invoice line item text array. |
| Starting at N+1 | One line descriptions of each of the elements that the invoice line item array, passed in the BLOB variable, is expected to contain. |
| These variables are also stuffed into the blob vBT1Blob which can be read on the 4D Open client. | |

The characters in vBTErrorText before the first carriage return give the size of the invoice header array that is expected. The characters in vBTErrorText before the second carriage return give the size of the invoice line item array that is expected.

These values can be extracted by finding the position of the first or second carriage return, extracting the characters lying before them, and converting them to numbers.

Two heading lines follow. After these are one-line descriptions of the contents of each of the invoice header and line item array elements. This description is placed in carriage return delimited format. That is, each element of the array is described using a separate line in the vBTErrorText variable.

The values in the arrays are also packed into the blob vBT1Blob which can be read on the 4D Open client side. The arrays can be extracted and their sizes determined using the Size of Array command.

All of the invoice data will need to be placed in text arrays before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text arrays then need to be packed into a BLOB. This BLOB is passed as $3 when the call is made to create the invoice. Refer to the following section “Inserting a new invoice” on page 129 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

50
11
50 element invoice header text array
11 element invoice line item text array
Invoice ID
Customer ID
Warehouse ID
Salutation, billing
First Name, billing
Last Name, billing
Company Name, billing
Address A, billing
Address B, billing
Address C, billing
City, billing
State, billing
Postal Code, billing
Zip Plus 4, billing
Country, billing
Email, billing
Salutation, shipping
First Name, shipping
Last Name, shipping
Company Name, shipping
Address A, shipping
Address B, shipping
Address C, shipping
City, shipping
State, shipping
Postal Code, shipping
Zip Plus 4, shipping
Country, shipping
Email, shipping
Concatenated Tax ID's, each followed by '/'
Due date
Order date

Ship date, for current shipment
Cancel date
Request date
Shipping amount
Order amount paid
Miscellaneous charge
Invoice type: 'stock' or 'service'
Invoice code
Sales terms including days until due and sales discount percent.~
Must be in 4Q terms format.
Shipping carrier for current shipment
Comment
Credit card number
Credit card expiration
Credit card type
Cash payment method
Customer PO
Tracking code for current shipment
Shipping priority for current shipment
Trade discount amount
Tax on shipping (1=yes)
_____ line item _____
Inventory item ID
Quantity ordered
Quantity shipped
Unit price
if “1” then taxed, otherwise not taxed
Comment 1
Comment 2
Comment 3
Serial number
Warehouse ID
Line item text

Execute the “Invoice_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

## Checking Data

### Checking the data supplied with a new invoice

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new invoice record described in the BLOB satisfies 4Q's invoice entry requirements.

This checking requires:

- the invoice information be properly formatted in the BLOB, and
- the data supplied is consistent with 4Q's invoice entry requirements, as are described below.
- the quantities *shipped* of each specified item be available in inventory. That is, attempting to ship more items than are in stock will return an error and the invoice will not be entered.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- it *does not* determine whether or not the invoice record or records related to the invoice, such as accounts, inventory, and shipping records, can be immediately created.
- it *does not* lock any records in anticipation for the actual submission of new invoice information.
- it *does not* determine if there are sufficient quantities *available* to satisfy the as specified in the invoice line items. This is because an invoice can still be accepted when the quantity ordered exceeds the quantity available.

Getting confirmation from this task does not guarantee that the invoice can be created or modified immediately.

Other concurrent processes may have access to needed counters, accounts or other records at the time when the invoice data is eventually submitted.

If such record locks occur when the invoice is submitted the server will cache the invoice data until updating can be done. This is described further in the section dealing with batch task processing (See "Brief Overview of the Batch Task System" on page 3).

To start this task for the creation of a new invoice call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "INVOICE_NEW_CHECK"

$3= Full invoice data packed in the BLOB (formatted as described below).

To start this task for the modification of an existing invoice use the parameters.

$1= Application, user, and method name in a text string.
$2= “INVOICE_MODIFY_CHECK”
$3= Full invoice data packed in the BLOB (formatted as described below).

The data passed in the BLOB must first be placed in a text array of the correct size. The invoice data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. This code represents an invoice with 2 lines. This example implies, but does not explicitly show, the assignment of all the intermediate array elements

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyInvoiceHeaderArray;49)
$MyInvoiceHeaderArray{1}:="0"
  ...
$MyInvoiceHeaderArray{49}:="Standard" `sample shipping priority
VARIABLE TO BLOB($MyInvoiceHeaderArray; $DataBlob;*)

ARRAY TEXT($MyLineItemArray;10)
MyLineItemArray{1}:="425" `sample item ID
...
MyLineItemArray{10}:="123456" `sample serial number
VARIABLE TO BLOB($MyInvoiceHeaderArray; $DataBlob; *)
MyLineItemArray{1}:="-1" `passing -1 identifies this line as “text only”
...
MyLineItemArray{10}:="" `no serial number
VARIABLE TO BLOB($MyInvoiceHeaderArray; $DataBlob; *)
```

The current invoice data structure is given in the following table. The table also provides notes regarding the use or formatting of the data that is supplied.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases one item or another item is required. This is described further in the “Description” column.

Not all fields in the invoice table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

Invoice line items are specified in the line item arrays that are packed in the BLOB. Each line item array must either specify a valid inventory item ID, or provide an ID value of 0. If zero is provided, then the line item is considered to be of text-type and no price, order or ship quantity are accepted. Text-type line items are only for annotating the invoice.

A complete description of the invoice fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the invoice data structure expected in the current version of 4th Quarter that you are communicating with you should execute the "Invoice_Description" call.

**TABLE 33.** Invoice Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| *Invoice Header array elements* | | | |
| 1 | X[0] | Invoice ID | [0]Not used for a new invoice. Must be the actual > 0 value for an existing invoice. |
| 2 | X | Customer ID | Must be > 0. |
| 3 | | Warehouse ID | If value >0 then must match existing Warehouse record in 4Q. Supply a value <= 0 if unused. In this case the value -1 is assigned. |

**TABLE 33.** Invoice Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 4 | | Salutation, billing | Default billing info stored with the indicated customer is used if the value “*” is supplied for a given field. |
| 5 | | First Name, billing | |
| 6 | | Last Name, billing | |
| 7 | | Company billing | |
| 8 | | Address A, billing | |
| 9 | | Address B, billing | |
| 10 | | Address C, billing | |
| 11 | | City, billing | |
| 12 | | State, billing | |
| 13 | | Postal Code, billing | |
| 14 | | Zip Plus 4, billing | |
| 15 | | Country, billing | |
| 16 | | Email, billing | |
| 17 | | Salutation, shipping | Default shipping info stored with the indicated customer is used if the value “*” is supplied for a given field. |
| 18 | | First Name, shipping | |
| 19 | | Last Name, shipping | |
| 20 | | Company Name, shipping | |
| 21 | | Address A, shipping | |
| 22 | | Address B, shipping | |
| 23 | | Address C, shipping | |
| 24 | | City, shipping | |
| 25 | | State, shipping | |
| 26 | | Postal Code, shipping | |
| 27 | | Zip Plus 4, shipping | |
| 28 | | Country, shipping | |
| 29 | | Email, shipping | |
| 30 | | Concatenated Tax ID's | Each ID must be followed by '/'. |

**TABLE 33.** Invoice Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 31 | | Due date | All dates should formatted as '##/##/####' |
| 32 | X | Order date | |
| 33 | X[1] | Ship date | [1]Applies to current shipment. Ship date is only required if items are marked as having been shipped. |
| 34 | | Cancel date | If the order cannot be partially shipped by this date it should be canceled. |
| 35 | | Request date | Date at which the customer would like to receive shipment. |
| 36 | | Shipping amount | Cost for current shipment only. |
| 37 | | Order amount paid | Amount paid, or considered paid, at time the order is entered. This is debited to 4Q's default invoice-related cash account. |
| 38 | | Miscellaneous charge | |
| 39 | X | Invoice type | 'stock' or 'service' |
| 40 | X | Invoice code | |
| 41 | | Sales terms | Must be in 4Q terms format: "Type nnn ppp ddd" where Type should be from 4Q's list "TypesOfTerms", nnn is # days net, ppp is sales % discount, ddd is days for sales discount. |
| 42 | | Shipping carrier for current shipment | Should be in 4Q's "Shipping Carriers" list for consistency. |
| 43 | | Comment | |
| 44 | | Credit card number | Containing numeric characters only. |
| 45 | | Credit card expiration | Must be in format ##/####. |
| 46 | | Credit card type | Any value accepted, but it should be an item from 4Q's "CreditCards" list for consistency. |
| 47 | | Cash payment method | Should be contained in 4Q's "Cash Payment" list for consistency. |
| 48 | | Customer PO | |
| 49 | | Tracking code for current shipment | |
| 50 | | Shipping priority for current shipment | should be contained in 4Q's "Shipping Priority" list for consistency. |

**TABLE 33.** Invoice Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 51 | | Sales person ID | If >0 then must match the ID of sales rep in the 4Q [SalesPerson] table. Supply a value <= 0 if unused. In this case the value -1 is assigned. |
| 52 | | Trade discount amount | Must be >=0 if supplied. The trade discount percent is calculated from this. Trade discount amounts are subtracted before sales tax. |
| 53 | | Tax on shipping (1=yes) | Includes shipping amount in computation of sales tax. Only the value "1" indicates "yes". |
| 54 | | Integer value indicating the account to which sales credit will be directed. | If "1" then sales are directed to the customer's general sales account.<br>If "2" then sales are directed to the customer's taxable sales account.<br>If "3" then sales are directed to each inventory item's assigned sales account, and miscelaneous charges and shipping are credited to the customer's general sales account.<br>Or else if any other value is passed then the system will used the default sales account assignment set by the administrator. |
| 55 | | Invoice's internal memo. | Text value assigned to the invoice which is not printed on the invoice that is created for the customer. |
| *Line Item Array Elements* | | | |
| 1 | X[2] | Inventory item ID | [2]If #0 then this must correspond to an existing item. If ID=-1, then this is treated as a text-only line with no cost or impact on inventory levels. Zero and other negative ID value are not accepted. Specification of "*" not allowed when modifying line items. |
| 2 | | Quantity ordered | Must be >= 0 |
| 3 | | Quantity to ship now | Must be >= 0 |
| 4 | | Unit price | |
| 5 | | 1=taxed, otherwise not taxed | |
| 6 | | Comment 1 | User specified text field. |
| 7 | | Comment 2 | User specified text field. |

**TABLE 33.** Invoice Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 8 | | Comment 3 | Alpha 30 field. |
| 9 | | Serial number | Alpha 30 field. |
| 10 | | Warehouse ID | Warehouse used for this particular item. |
| 11 | | Line item text | Used only for text-type lines where Invent. ID is given as -1, see array element 1 above. |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.

**TABLE 34.** Invoice Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -105 | Invoice ID <=0. |
| -110 | Customer ID <= 0. |
| -115 | Customer ID= <> not found. |
| -118 | Invoice with ID = <…> could not be found. |
| -120 | Customer ID= <> has no receivable account. |
| -125 | Customer ID= <> has no sales account. |
| -129 | Invoice type given as <>. Must be either 'stock' or 'service'. |
| -130 | Customer ID= <> missing tax or nontax sales account. |
| -132 | Customer ID= <> missing billing name and company name. |
| -133 | Customer ID= <> has no billing street address. |

**TABLE 34.** Invoice Data Value Error

| Error Code | Description |
|---|---|
| -134 | Customer ID= <> missing billing city and state. |
| -135 | Customer ID= <> missing shipping name and company name. |
| -140 | Customer ID= <> has no shipping street address. |
| -145 | Customer ID= <> missing shipping city and state. |
| -146 | Trade Discount= <> must be >= 0. |
| -150 | Sales tax ID <=0. |
| -155 | Sales tax ID= <> cannot be found. |
| -160 | Sales tax ID= <> has no related account. |
| -165 | Account ID = <> for Sales tax ID= <> cannot be found. |
| -168 | Invoice due date cannot be set until some items are shipped. |
| -170 | Invoice due date is required when some items are shipped. |
| -175 | Missing invoice order date. |
| -180 | Negative invoice shipping amount. |
| -185 | Negative order amount paid. |
| -190 | Missing invoice code. |
| -195 | Duplicate code: invoice with code <> already exists. |
| -200 | Terms of sale = <> are given in an unrecognizable format. |
| -201 | Unspecified problem handling Terms of sale = <> |
| -203 | Sales person with ID = <> cannot be found.<br>(This value is only tested if a value >0 is provided) |
| -204 | Invoice warehouse with ID = <> cannot be found.<br>(This value is only tested if a value >0 is provided) |
| -205 | Line item number <> contains <> rather than <> columns. |
| -208 | Line number <> gives bad item ID = <>. |
| -210 | Line item with ID= <> has quantity shipped (= <>) greater than quantity ordered (= <>). |
| -215 | Line with item ID= <> cannot be found. |

**TABLE 34.** Invoice Data Value Error

| Error Code | Description |
|---|---|
| -220 | Line with item ID= <> has quantity shipped (= <>) greater than quantity available (= <>). |
| -210 | Invoice ID could not be obtained or could not be assigned. |
| -230 | Invoice record with ID = <…> cannot be deleted. |
| -235 | Invoice record with ID = <…> is locked. |

# New Invoice

## Inserting a new invoice

A call to _BT4QSubmitTask will attempt to insert a new invoice record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “INVOICE_NEW”
$3= Full invoice data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Invoice_New_Check”. It is often preferable to be able to check the invoice data before submitting it as the 4D Open will have more control over exception processing.

Errors detected when the invoice is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values returned if the invoice data is not acceptable, shown in Table 33, the following error code values may be returned if the system is not able to process the request to insert the invoice.

**TABLE 35.** Insert Invoice Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new invoice ID could not be obtained. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Invoice data is unacceptable. |
| value >0 | The new ID value to be assigned to the invoice. |

## Batch Task Processing

The call to insert an invoice is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record before it returns a positive invoice ID value. Rather, it

confirms the invoice data, obtains the ID values that it will need, and creates a Batch Task record.

The actual insertion of new data into the invoice and related tables begins after the new invoice's ID value is returned to the 4D Open client.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 33. If the data does not pass these tests, as can be determined ahead of time by using the "Invoice_New_Check" call, then an error is returned and no batch task record is created.

Even though the supplied data is accepted, the system may not be able to enter the invoice record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new invoice for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to an administrator-defined maximum number of attempts.

If the system cannot successfully process the invoice insertion request after the maximum number of attempts has been reached, then the insert invoice Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. Reused task records are assigned new ID values.

## Invoice Batch Processing

4th Quarter handles new invoices one of two ways depending on how the administrator has setup the data base.

- Journalizing in Batch Mode

  When invoice entries are set for batch processing the system will save the invoice and update the inventory, but will not create an accounting transaction at that time. Accounting transactions are created later when the all pending, unjournalized invoices are processes in a batch.

- Journalizing in Real-time

  When invoices are set to enter in real-time the system saves the invoice, updates the inventory, and creates an accounting transaction. The invoices are journalized at this time and account balances are brought up to date.

When invoices are created through the API's 4th Quarter continues to either process the invoices in real-time, or hold them in a batch to be journalized later. This invoice batch setting is assigned by the administrator on the invoice page of the Maintenance screen.

The batching of invoices has no relation to the "batch task" records used to store tasks submitted to 4th Quarter through the API's. For example, if invoices are set to be journalized in batch, then whether the submitted invoice is created or not it will not be journalized.

All the information needed to later journalize the invoice is stored in the invoice itself, and the invoice will be journalized in the same manner as invoices created from within 4Q. That is, it will be journalized when the invoice batch is processed.

## Modify Invoice

### Modifying an existing invoice

A call to _BT4QSubmitTask will attempt to modify an existing invoice record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “INVOICE_MODIFY”
$3= Full invoice data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Invoice_Modify_Check”. It is often preferable to check the invoice data before submitting it for application to the invoice. The caller will have more control over exception processing in this case.

The data submitted to describe the modified invoice must include all invoice information. If there are certain fields of the invoice that you do not want to modify then you can pass the asterisk as a text array element. This only applies to values that are not required for locating the invoice and for data that appears in the invoice body. It does not apply to any of the line items values.

All line item values for a modified invoice must be fully specified. The line items of the existing invoice are completely removed. Their effects on inventory and accounting are reversed. The line items of the modified invoice are completely replaced with the line items specified in the API call.

The most obvious difference between the data supplied when modifying an invoice versus adding an invoice is that the first element in the invoice header data array must contain the ID of the invoice to be modified. This must be a positive number passed as a string value in the text array that is contained within the invoice data BLOB.

If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values. Fields in the invoice body that are to be left unchanged must be assigned the asterisk character (“*”).

Errors detected when the invoice is submitted may be reported differently from when the data is submitted for testing. This is because updating a record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the invoice data is not acceptable, the following error code values may be returned if the system is not able to process the request to update the invoice.

**TABLE 36.** Insert Invoice Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-4” | The invoice with the specified ID could not be located. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Invoice data is unacceptable. |
| value >0 | The new ID value to be assigned to the invoice. |

## Batch Task Processing

The call to update an invoice is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the existing record before it returns a positive invoice ID value. Rather, it confirms the invoice data and created a Batch Task record The actual table update begins just after the confirming invoice ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to update the invoice record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new invoice for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to a administrator-defined maximum number of attempts.

If the system cannot successfully process the invoice update request after the maximum number of attempts has been reached, then the update invoice Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,

- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the 4D Open page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. New task ID's are assigned when records are overwritten.

## Batch Journalizing

If an existing invoice has been journalized, then any modifications to it are immediately processed to alter the balances of related accounts. Batch journalizing of invoices only applies to new invoices that have not been journalized.

On the other hand, if the invoice being modified has not yet been journalized, then the API call to modify the invoice will either leave the changed invoice unjournalized or it will cause the system to journalize the invoice upon saving the changes. The action taken depends upon whether the 4Q system has Invoice Batch Processing (where "batch processing" here refers to "batch journal processing") turned on or off.

# Delete Invoice

## Deleting an invoice

To delete an invoice from the 4Q data file call _BT4QSubmitTask with the action code “INVOICE_DELETE”. The BLOB must contain a text array that has at least one element. The first element of this array must contain the ID of the ID of the invoice to be erased. No other array values are referenced.

The following error codes may be returned in the vBTErrorCode variable.

**TABLE 37.** Delete Invoice Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Invoice deletion error, refer to message in vBTErrorText. |
| “-1” | Unrecognized action parameter. |
| “-2” | No invoice exists to match the supplied invoice ID. |
| “-3” | The invoice ID cannot be recognized. Check its value and format. |
| “-4” | The invoice is locked and cannot be deleted. |
| "-6" | Changes to any 4Q tables are currently forbidden. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 8

# PO Data Entry Methods

Update tasks supported by the 4Q API's affecting the "vPurchaseOrder" table.

4th Quarter supports the following invoice-related tasks through the _BT4QSubmitTask method. Each of these tasks is done through a 4D Open call to the _BT4QSubmitTask method.

Each task passes a different "action" parameter and, according to the task, BLOB of data.

**TABLE 38.** Purchase Order Table Operations

| Action Parameter | Description |
|---|---|
| Purchase_Description | Returning a description of the purchase order data format. |
| Purchase_New_Check | Checking the data supplied for a new purchase order. |
| Purchase_Modify_Check | Checking the data supplied for the modification of an existing purchase order. |
| Purchase_New | Inserting a new purchase order. |
| Purchase_Modify | Modifying an existing purchase order. |
| Purchase_Delete | Deleting an purchase order. |

Each task will different information back from the server. Details are described below.

## *Invoice Data Format*

### Returning a description of the purchase order data format

This task returns to the 4D Open client a description of the data that is expected to be passed to the server when the client requests the server create a new purchase order. This information is passed back in the text of the vBTErrorText variable.

To start this task call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “PURCHASE_DESCRIPTION”
$3= empty BLOB (data is not used)

Once the client has opened a 4D Open connection to the server and obtained a connection ID value use the following 4D code to submit the task:

```
C_Blob(vMyBlob)
C_Text(vIdentity; vAction; $Method; $ProcessName)
C_Longint($4DOpenErrCode; $Stack128K; $MyProcessID)
Set Blob Size(vMyBlob;0)
`Note: the value of MyConnectID is known from when the connection
                                    is opened.
vIdentity:=Application File+ ";" + Current User +
                                    <name of current method>
vAction:= "Purchase_Description"
$Method:="_BT4QSubmitTask"
$ProcessName:="TestPurchase"
$Stack128K:=128000
vBTErrorCode:=""
vBTErrorText:=""
vBlank:=""
v4QLockoutFlag:=""
$4QIsLocked:=False
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                    $MyProcessID; "<>vSYDB_LockoutFlag";
                    "v4QLockoutFlag")

If($4DOpenErrCode = 0)
      $4DOpenErrCode:=OP Set Semaphore (MyConnectID;
                    "<>vSYDB_LockoutFlag"; vSemaphoreState)
      $4QIsLocked:=vSemaphoreState
      $$4DOpenErrCode:=OP Clear Semaphore (MyConnectID;
                    "<>vSYDB_LockoutFlag")
End if
```

```
Case of
:($4DOpenErrCode # 0)
      ` Handle 4D Open error processing here.
:($4QIsLocked)
      ` Defer communications until 4Q global database lock is released.
Else
$4DOpenErrCode:=OP Execute On Server (MyConnectID;
      $Method; $Stack128K; $ProcessName;
      $MyProcessID; "vIdentity"; "vAction"; "vMyBlob")

If($4DOpenErrCode = 0)
      vMessage:=""
      $GiveUpTime:=Current time+(30)`wait 30 seconds max.
      $TicksToWait:=20 `delay between 4D Open calls to prevent
                  `flooding the server.
      $4DOpenErrCode:=OP Get Process Variable (MyConnectID;
            $MyProcessID;"vBTErrorText"; "vMessage")

      While ((vBTErrorCode = "") &
                        (Current time<$GiveUpTime))
            DELAY PROCESS(Current process;$TicksToWait)
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; "$MyProcessID";
                  "vBTErrorCodeCode";"vBTErrorCode")
            $4DOpenErrCode:=OP Get Process Variable (
                  MyConnectID; "$MyProcessID";"vBTErrorText";
                  "vMessage")
            $4DOpenErrCode:=OP Set Process Variable (
                  MyConnectID; $MyProcessID;"vBTErrorCode";
                  "vBlank")
End while

      $loc:=Position(char(Tab);vBTErrorCode)
      If($loc>1)
            $BTErrorNumber:=Num(Substring
                  (vBTErrorCode;1;$loc-1))
      Else
            $BTErrorNumber:=Num(vBTErrorCode)
      End if

      Case of
```

```
:($BTErrorNumber<0)
      ` Server has returned an error.
      ` Take appropriate response.
Case of
:(Num(vBTErrorCode)<0)
      ` Server has returned an error.
      ` Take appropriate response.
:(vBTErrorCode="")
      ` Timed-out waiting for a response.
      ` Process error here.
Else
      $PurchaseDataSequence:=vMessage
      ` Task completed successfully.
      ` Display purchase order data text.
End case
End case
```

The value returned in vSemaphoreState indicates whether the 4Q database is currently locked to all updates. See “Global Lock-Outs” on page 167 for more details.

The value returned in the vBTErrorCode variable determines whether the call was processed successfully on the server side. It is returned by the server in the following format:

```
vBTErrorCode =
      result code(integer) + Tab +{“TaskID=” + <Task ID> +Tab} +
      current date +Tab + current time
```

The {“TaskID=” + <Task ID> +Tab} portion is only included if a batch task record has been created.

It returns the following result code values.

- Number value < 0 (when the string is converted to a number): error condition. In this case the task ID, date and time are not returned.
- Empty string: processing has not begun.
- Invoice ID: request has been accepted and is being processed.

The value returned in the vBTErrorText variable describes the invoice data that 4th Quarter expects to receive as elements in multiple text arrays when a new purchase order is created through a 4D Open call.

The structure of the vBTErrorText message is as follows:

**TABLE 39.** Structure of the Purchase Order Text Description in vBTErrorText

| Line # | Content |
|---|---|
| 1 | The number of elements in the required purchase order header text array. |
| 2 | The number of elements in each of the required purchase order line item text arrays. |
| 3 | A header identifying the following information as pertaining to purchase order header data. |
| 4 | A header identifying the following information as pertaining to purchase order line item data. |
| Starting at 5 | One line descriptions of each of the elements that the purchase order header text array, passed in the BLOB variable, is expected to contain. |
| N (depends on version) | At some number of lines down in the text there appears the string "_____ line item _____" as an marker that the following text pertains to the structure of the purchase order line item text array. |
| Starting at N+1 | One line descriptions of each of the elements that the purchase order line item array, passed in the BLOB variable, is expected to contain. |
| These variables are also stuffed into the blob vBT1Blob which can be read on the 4D Open client. | |

The characters in vBTErrorText before the first carriage return give the size of the purchase order header array that is expected. The characters in vBTErrorText before the second carriage return give the size of the purchase order line item array that is expected.

These values can be extracted by finding the position of the first or second carriage return, extracting the characters lying before them, and converting them to numbers.

Two heading lines follow. After these are one-line descriptions of the contents of each of the purchase order header and line item array elements. This description is placed in carriage return delimited format. That is, each element of the array is described using a separate line in the vBTErrorText variable.

The values in the arrays are also packed into the blob vBT1Blob which can be read on the 4D Open client side. The arrays can be extracted and their sizes determined using the Size of Array command.

All of the purchase order data will need to be placed in text arrays before being packed into a BLOB. This means that when numbers or dates are required, they must be converted to string values in order to be placed in the elements of the text array.

The text arrays then need to be packed into a BLOB. This BLOB is passed as $3 when the call is made to create the purchase order. Refer to the following section “Inserting a New Purchase Order” on page 154 for details on packing the array into the BLOB.

As an example, at the time of this writing the value returned by vBTErrorText is the following. This example may not be the value actually returned by a subsequent version of 4th Quarter.

58
11
58 element purchase order header text array
11 element purchase order line item text array
Purchase order ID
Vendor ID
Warehouse ID
Salutation, billing
First Name, billing
Last Name, billing
Company Name, billing
Address A, billing
Address B, billing
Address C, billing
City, billing
State, billing
Postal Code, billing
Zip Plus 4, billing
Country, billing
Email, billing
Salutation, receiving
First Name, receiving
Last Name, receiving
Company Name, receiving
Address A, receiving
Address B, receiving
Address C, receiving
City, receiving
State, receiving
Postal Code, receiving
Zip Plus 4, receiving
Country, receiving
Email, receiving
PO approved (0=no, 1=yes)
Concatenated Tax ID's, each followed by '/'
Due date

Order date
Ship date, for current shipment
Cancel date
Request date
Shipping amount
Order amount paid
Miscellaneous charge
ID of cash account where payments are drawn, -1 for system default
Purchase order type: 'stock' or 'service'
Purchase order code (alpha numeric)
Sales terms including days until due and sales discount percent.~
Must be in 4Q terms format.
Shipping carrier for current shipment
Comment
Cash payment method
Vendor invoice number
Vendor voice phone number
Vendor fax phone number
Tracking code for current shipment
Shipping priority for current shipment
Name of person who placed the order
Name of person who approved the order
Name of vendor contact person
Trade discount amount
Tax on shipping (0=no, 1=yes)
Credit Inventory Purchase accounts (1=yes), or vendor purchase account (blank)
Memo printed on PO
Internal memo, not printed on PO
_____ line item _____
Inventory item ID
Quantity ordered
Quantity received
Unit price
if “1” then taxed, otherwise not taxed
Comment 1
Comment 2
Comment 3
Serial number
Warehouse ID
Line item text (used only if ID=0)

Execute the “Purchase_Description” call to receive the current data structure for the version of 4th Quarter that you are working with.

# Checking Data

## Checking the Data Supplied with a New Purchase Order

This task examines the data contained in a BLOB passed to it as a parameter in the call. The task returns an error code that indicates whether the new purchase order record described in the BLOB satisfies 4Q's purchase order entry requirements.

This checking requires:

- the purchase order information be properly formatted in the BLOB, and
- the data supplied is consistent with 4Q's invoice entry requirements, as are described below.

This task performs "read only" operations. There are some aspect of new record creation that this task does not do:

- it *does not* determine whether or not the purchase order record or records related to the purchase order , such as accounts, inventory, and shipping records, can be immediately created.
- it *does not* lock any records in anticipation for the actual submission of new purchase order information.

Getting confirmation from this task does not guarantee that the purchase order can be created or modified immediately.

Other concurrent processes may have access to needed counters, accounts or other records at the time when the purchase order data is eventually submitted.

If such record locks occur when the purchase order is submitted the server will cache the purchase order data until updating can be done. This is described further in the section dealing with batch task processing (See "Brief Overview of the Batch Task System" on page 3).

To start this task for the creation of a new invoice call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name in a text string.
$2= "PURCHASE_NEW_CHECK"
$3= Full purchase order data packed in the BLOB (formatted as described below).

To start this task for the modification of an existing purchase order use the parameters.

$1= Application, user, and method name in a text string.

$2= “PURCHASE_MODIFY_CHECK”

$3= Full purchase order data packed in the BLOB (formatted as described below).

The data passed in the BLOB must first be placed in a text array of the correct size. The purchase order data must be placed in the array elements in the correct order.

Assigning elements to a text array and placing a text array in a BLOB can be done using the following 4D code. This code represents an purchase order with 2 lines. This example implies, but does not explicitly show, the assignment of all the intermediate array elements

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($MyPurchaseHeaderArray;49)
$MyPurchaseHeaderArray{1}:="0"
  ...
$MyPurchaseHeaderArray{49}:="Standard" `sample shipping priority
VARIABLE TO BLOB($MyPurchaseHeaderArray; $DataBlob;*)

ARRAY TEXT($MyLineItemArray;10)
MyLineItemArray{1}:="425" `sample item ID
...
MyLineItemArray{10}:="123456" `sample serial number
VARIABLE TO BLOB($MyPurchaseHeaderArray; $DataBlob; *)
MyLineItemArray{1}:="-1" `passing -1 identifies this line as “text only”
...
MyLineItemArray{10}:="" `no serial number
VARIABLE TO BLOB($MyPurchaseHeaderArray; $DataBlob; *)
```

The current purchase order data structure is given in the following table. The table also provides notes regarding the use or formatting of the data that is supplied.

An “X” appears in the “Required” column when the corresponding data must be supplied. The X is superscripted in those cases one item or another item is required. This is described further in the “Description” column.

Not all fields in the purchase order table are assigned values through this call. Some fields are determined by 4Q through reference to existing, related data. Items whose values depend on submitted data are calculated by 4th Quarter.

Purchase order line items are specified in the line item arrays that are packed in the BLOB. Each line item array must either specify a valid inventory item

ID, or provide an ID value of 0. If zero is provided, then the line item is considered to be of text-type and no price, order or ship quantity are accepted. Text-type line items are only for annotating the purchase order.

A complete description of the purchase order fields, data types, and formatting requirements is located in the 4th Quarter Data Dictionary. This document is available on-line to licensors of 4th Quarter, and for others is available by request from Braided Matrix.

This data structure may change in future versions of 4th Quarter. In order to determine the purchase order data structure expected in the current version of 4th Quarter that you are communicating with you should execute the "Purchase_Description" call.

**TABLE 40.** Purchase Order Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| *Purchase Order Header array elements* | | | |
| 1 | X[0] | Purchase Order ID | [0]Not used for a new purchase order . Must be the actual > 0 value for an existing purchase order. |
| 2 | X | Vendor ID | Must be > 0. |
| 3 | | Warehouse ID | If value >0 then must match existing Warehouse record in 4Q. Supply a value <= 0 if unused. In this case the value -1 is assigned. |

**TABLE 40.** Purchase Order Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 4 | | Salutation, billing | Default billing info stored with the indicated customer is used if the value "*" is supplied for a given field.<br><br>A billing name and/or company name must either be on file with the vendor, or supplied with the PO.<br><br>A billing street address must either be on file with the vendor or supplied with the PO.<br><br>A billing city or state must either be on file with the vendor or supplied with the PO. |
| 5 | | First Name, billing | |
| 6 | | Last Name, billing | |
| 7 | | Company billing | |
| 8 | | Address A, billing | |
| 9 | | Address B, billing | |
| 10 | | Address C, billing | |
| 11 | | City, billing | |
| 12 | | State, billing | |
| 13 | | Postal Code, billing | |
| 14 | | Zip Plus 4, billing | |
| 15 | | Country, billing | |
| 16 | | Email, billing | |
| 17 | | Salutation, receiving | Default receiving info assigned by the administrator in the "Your Addresses" area on the System page of the Admin Setting screen. Default address information is used if the value "*" is supplied for a given field..<br><br>Your receiving name and/or company name must either be on file or supplied with the PO. |
| 18 | | First Name, receiving | |
| 19 | | Last Name, receiving | |
| 20 | | Company Name, receiving | |
| 21 | | Address A, receiving | |
| 22 | | Address B, receiving | |
| 23 | | Address C, receiving | |
| 24 | | City, receiving | |
| 25 | | State, receiving | |
| 26 | | Postal Code, receiving | |
| 27 | | Zip Plus 4, receiving | |
| 28 | | Country, receiving | |
| 29 | | Email, receiving | |
| 30 | | PO Approved (1=yes) | |

**TABLE 40.** Purchase Order Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 31 | | Text code or concatenated Tax ID's | Pass “” (blank) to add no tax, “VENDOR” to indicate use of tax accounts assigned to the vendor, “DEFAULT” to use the purchase order defaults stored in the system, or specifiy individual tax accounts by specifying their ID values followed by '/'. |
| 32 | X[1] | Due date | All dates should formatted as '##/##/####'. Due date isrequired if items have been received. |
| 33 | X | Order date | |
| 34 | X[1] | Received date | [1]Applies to current shipment. Receive date is required if items have been received. |
| 35 | | Cancel date | If the order cannot be partially shipped by this date it should be canceled. |
| 36 | | Request date | Date at which you would like to receive shipment. |
| 37 | | Shipping amount | Cost for current shipment only. Must be >=0. |
| 38 | | Order amount paid | Amount paid, or considered paid, at time the order is entered. Must be >=0. |
| 39 | | Miscellaneous charge | Must be >=0. |
| 40 | | ID of cash account to draw from | Use -1 for the system’s default PO cash account. |
| 41 | X | Purchase Order type | 'stock' or 'service' |
| 42 | X | Purchase Order code | Alpha numeric, must be unique. Leading and trailing spaces will be removed. |
| 43 | | Sales terms | Must be in 4Q terms format: “Type nnn ppp ddd” where Type should be from 4Q’s list “TypesOfTerms”, nnn is # days net, ppp is sales % discount, ddd is days for sales discount. |
| 44 | | Shipping carrier for current shipment | Should be in 4Q's “Shipping Carriers” list for consistency. |
| 45 | | Cash payment method | Should be contained in 4Q's “Cash Payment” list for consistency. |
| 46 | | Vendor invoice number | |
| 47 | | Vendor voice phone number | |

**TABLE 40.** Purchase Order Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 48 | | Vendor fax phone number | |
| 49 | | Tracking code for current shippment | |
| 50 | | Shipping priority for current shipment | |
| 51 | | Name of person who placed the order | |
| 52 | | Name of person who approved the order | |
| 53 | | Name of vendor contact person | |
| 54 | | Trade discount amount | Must be >=0 if supplied. The trade discount percent is calculated from this. Trade discount amounts are subtracted before sales tax. |
| 55 | | Tax on shipping (1=yes) | Includes shipping amount in computation of sales tax. Only the value “1” indicates “yes”. |
| 56 | | Integer value indicating the account to which sales credit will be directed. | If left blank, then purchases are directed to the vendor’s general purchase account.<br>If “1” then purchases are directed to each inventory item’s assigned purchase account, and miscelaneous charges and shipping are credited to the vendor’s general purchase account. |
| 57 | | Memo printed on PO | |
| 58 | | Internal memo, not printed on PO | Text value assigned to the purchase order which is not printed on the purchase order that is created for the vendor. |
| *Line Item Array Elements* | | | |
| 1 | X[2] | Inventory item ID | [2]If #0 then this must correspond to an existing item. If ID=-1, then this is treated as a text-only line with no cost or impact on inventory levels. Zero and other negative ID value are not accepted. Specification of “*” not allowed when modifying line items. |
| 2 | | Quantity ordered | Must be >= 0 |
| 3 | | Quantity received | Must be >= 0 |
| 4 | | Unit price | |
| 5 | | 1=taxed, otherwise not taxed | |
| 6 | | Comment 1 | User specified text field. |

**TABLE 40.** Purchase Order Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 7 | | Comment 2 | User specified text field. |
| 8 | | Comment 3 | Alpha 30 field. |
| 9 | | Serial number | Alpha 30 field. |
| 10 | | Warehouse ID | Warehouse used for this particular item. |
| 11 | | Line item text | Used only for text-type lines where Invent. ID is given as -1, see array element 1 above. |

If the data supplied in the array elements does not satisfy entry requirements, or if there is an error in the format of the call itself, then an indication of this is passed back in the vBTErrorText variable.

- If the error is in the format of the call itself, then the method stops immediately and returns an error code that is a negative number.
- If the error is in the data values, then the task will continue past the first error to test all the supplied data.

  If this occurs, then the vBTErrorText variable will provide a ***description of all*** the format or value errors encountered. The following table lists the formatting errors that are reported.
- In the following table the symbol “<>” represents the value supplied in the API call.

**TABLE 41.** Purchase Order Data Value Error

| Error Code | Description |
|---|---|
| -30 | Unable to extract the data passed in BLOB parameter. BLOB may have been packed incorrectly. |
| -105 | Purchase Order ID <=0. |
| -110 | Vendor ID <= 0. |
| -118 | Vendor with ID = <…> not found. |
| -120 | Vendor ID = <…> has no payable accounts. |
| -125 | Vendor ID = <…> has no purchase account. |
| -129 | Purchase Order type given as <>. Must be either 'stock' or 'service'. |
| -132 | Vendor ID = <…> missing billing name and company name. |

**TABLE 41.** Purchase Order Data Value Error

| Error Code | Description |
|---|---|
| -133 | Vendor ID = <…> has no billing street address. |
| -134 | Vendor ID = <…> missing billing city and state. |
| -135 | Missing receiving name and company name. |
| -146 | Trade Discount = <…> must be >= 0. |
| -150 | Sales tax ID <=0. |
| -155 | Sales tax ID = <…> cannot be found. |
| -160 | Sales tax ID = <…> has no related account. |
| -165 | Account ID = <…> for Sales tax ID= <> cannot be found. |
| -170 | Purchase Order due date is required when some items are received. |
| -172 | Purchase Order received date is required when some items are received. |
| -175 | Missing purchase order order date. |
| -180 | Negative purchase order shipping amount. |
| -185 | Negative order amount paid. |
| -190 | Missing purchase order code. |
| -195 | Duplicate code: purchase order with code <> already exists. |
| -200 | Terms of sale = <…> are given in an unrecognizable format. |
| -201 | Unspecified problem handling Terms of sale = <…> |
| -203 | Sales person with ID = <…> cannot be found.<br>(This value is only tested if a value >0 is provided) |
| -204 | Purchase order warehouse with ID = <…> cannot be found.<br>(This value is only tested if a value >0 is provided) |
| -205 | Line item number <…> contains <…> rather than <…> columns. |
| -208 | Line number <…> gives negative item ID = <…>. |
| -210 | Line item with ID= <…> has quantity received (= <…>) greater than quantity ordered (= <…>). |
| -215 | Line with item ID = <…> cannot be found. |
| -210 | Purchase order ID could not be obtained or could not be assigned. |
| -230 | Purchase order record with ID = <…> cannot be deleted. |

**TABLE 41.** Purchase Order Data Value Error

| Error Code | Description |
|---|---|
| -235 | Purchase order record with ID = <…> is locked. |

## New Purchase Order

### Inserting a New Purchase Order

A call to _BT4QSubmitTask will attempt to insert a new invoice record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “PURCHASE_NEW”
$3= Full purchase order data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Purchase_New_Check”. It is often preferable to be able to check the purchase order data before submitting it as the 4D Open will have more control over exception processing.

Errors detected when the purchase order is submitted may be reported differently from when the data is submitted for testing. This is because inserting a new record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values returned if the purchase order data is not acceptable, shown in Table 41, the following error code values may be returned if the system is not able to process the request to insert the purchase order .

**TABLE 42.** Insert Purchase Order Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-2” | A new purchase order ID could not be obtained. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Purchase Order data is unacceptable. |
| value >0 | The new ID value to be assigned to the purchase order. |

### Batch Task Processing

The call to insert an purchase order is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the insertion does not actually add the new record before it returns a positive purchase order ID value. Rather, it confirms the purchase order data, obtains the ID values that it will need, and creates a Batch Task record.

The actual insertion of new data into the purchase order and related tables begins after the new purchase order's ID value is returned to the 4D Open client.

This batch task stage of processing only happens if the submitted data passes the data entry conditions given above in Table 40 and Table 41. If the data does not pass these tests, as can be determined ahead of time by using the "Purchase_New_Check" call, then an error is returned and no batch task record is created.

Even though the supplied data is accepted, the system may not be able to enter the purchase order record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new purchase order for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to an administrator-defined maximum number of attempts.

If the system cannot successfully process the purchase order insertion request after the maximum number of attempts has been reached, then the insert purchase order Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,
- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the API page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. Reused task records are assigned new ID values.

## Purchase Order Batch Processing

4th Quarter handles new purchase order one of two ways depending on how the administrator has setup the data base.

- Journalizing in Batch Mode

  When purchase order entries are set for batch processing the system will save the purchase order and update the inventory, but will not create an accounting transaction at that time. Accounting transactions are created later when the all pending, unjournalized purchase orders are processed in a batch.

- Journalizing in Real-time

  When purchase orders are set to enter in real-time the system saves the purchase order, updates the inventory, and creates an accounting transaction. The purchase orders are journalized at this time and account balances are brought up to date.

When purchase orders are created through the API's 4th Quarter continues to either process the purchase orders in real-time, or hold them in a batch to be journalized later. This purchase order batch setting is assigned by the administrator on the purchase order page of the Maintenance screen.

The batching of purchase orders has no relation to the "batch task" records used to store tasks submitted to 4th Quarter through the API's. For example, if purchase orders are set to be journalized in batch, then whether the submitted purchase order is created or not it will not be journalized.

All the information needed to later journalize the purchase order is stored in the purchase order itself, and the purchase order will be journalized in the same manner as purchase orders created from within 4Q. That is, it will be journalized when the purchase order batch is processed.

# Modify Purchase Order

## Modifying an Existing Purchase Order

A call to _BT4QSubmitTask will attempt to modify an existing purchase order record when the following parameters are passed:

$1= Application, user, and method name in a text string.
$2= “PURCHASE_MODIFY”
$3= Full purchase order data packed in the BLOB (formatted as described above).

The data in the BLOB must satisfy the same requirements that are imposed when the call is made to “Purchase_Modify_Check”. It is often preferable to check the purchase order data before submitting it for application to the purchase order . The caller will have more control over exception processing in this case.

The data submitted to describe the modified purchase order must include all purchase order information. If there are certain fields of the purchase order that you do not want to modify then you can pass the asterisk as a text array element. This only applies to values that are not required for locating the purchase order and for data that appears in the purchase order body. It does not apply to any of the line items values.

All line item values for a modified purchase order must be fully specified. The line items of the existing invoice are completely removed. Their effects on inventory and accounting are reversed. The line items of the modified purchase order are completely replaced with the line items specified in the API call.

The most obvious difference between the data supplied when modifying a purchase order versus adding a purchase order is that the first element in the purchase order header data array must contain the ID of the purchase order to be modified. This must be a positive number passed as a string value in the text array that is contained within the purchase order data BLOB.

If you leave certain fields blank, then those fields will be set to blank in the modified record, if the system allows these fields to be assigned blank values. Fields in the purchase order body that are to be left unchanged must be assigned the asterisk character (“*”).

Errors detected when the purchase order is submitted may be reported differently from when the data is submitted for testing. This is because updating a record can encounter problems beyond what those simply having to do with data syntax or entry values.

In addition to the error code values that will be returned if the purchase order data is not acceptable, the following error code values may be returned if the system is not able to process the request to update the purchase order .

**TABLE 43.** Insert Purchase Order Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| “-1” | Unrecognized action parameter. |
| “-4” | The purchase order with the specified ID could not be located. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-9” | The submission has been canceled, the Batch Task record has been deleted. |
| value <0 | Purchase order data is unacceptable. |
| value >0 | The new ID value to be assigned to the purchase order. |

## Batch Task Processing

The call to update an purchase order is handled by the server as quickly as possible.

To enable an error code to be returned rapidly the update does not actually update the existing record before it returns a positive purchase order ID value. Rather, it confirms the purchase order data and created a Batch Task record The actual table update begins just after the confirming purchase order ID value is returned to the 4D Open client.

Even though the supplied data is accepted, the system may not be able to update the purchase order record. This can happen, for example, due to the locking of related records that are being used to satisfy other user requests.

If the system is not able to enter the new purchase order for reasons such as this, then the system will make another attempt to enter the data after a brief pause. The system will continue trying to save the information up to a administrator-defined maximum number of attempts.

If the system cannot successfully process the purchase order update request after the maximum number of attempts has been reached, then the update purchase order Batch Task record will be saved to the data file to await further manual processing.

The 4Q administrator can review these Batch Task records to determine:

- what jobs were submitted,

- when and by whom they were submitted, and
- the result of the systems attempt to process them.

The administrator can then

- print,
- delete, or
- resubmit the Batch Task records.

If the system does succeed in processing the task, a Batch Task record is still created.

However in the case of successful processing the Batch Task record is only saved in the 4Q database for a set number of days. This duration is set by the 4Q administrator on the 4D Open page of the Maintenance screen.

Once the record of a successful task reaches this set age it is marked for overwriting.

Subsequent 4D Open calls to run new tasks may overwrite the previous batch task record at that point. New task ID's are assigned when records are overwritten.

## Batch Journalizing

If an existing purchase order has been journalized, then any modifications to it are immediately processed to alter the balances of related accounts. Batch journalizing of purchase orders only applies to new purchase orders that have not been journalized.

On the other hand, if the purchase order being modified has not yet been journalized, then the API call to modify the purchase order will either leave the changed purchase order unjournalized or it will cause the system to journalize the purchase order upon saving the changes. The action taken depends upon whether the 4Q system has Purchase Order Batch Processing (where "batch processing" here refers to "batch journal processing") turned on or off.

## Delete Purchase Order

### Deleting a Purchase Order

To delete a purchase order from the 4Q data file call _BT4QSubmitTask with the action code “PURCHASE_DELETE”. The BLOB must contain a text array that has at least one element. The first element of this array must contain the ID of the ID of the purchase order to be erased. No other array values are referenced.

The following error codes may be returned in the vBTErrorCode variable.

**TABLE 44.** Delete Purchase Order Error Codes

| Error Code Str | Description |
|---|---|
| “” (blank) | Submission has not yet completed. |
| value <0 | Purchase order deletion error, refer to message in vBTErrorText. |
| “-1” | Unrecognized action parameter. |
| “-2” | No purchase order exists to match the supplied purchase order ID. |
| “-3” | The purchase order ID cannot be recognized. Check its value and format. |
| “-4” | The purchase order is locked and cannot be deleted. |
| "-6" | Changes to any 4Q tables are currently forbidden. |

If the deletion succeeds, then the vBTErrorCode variable returns the ID of the deleted record followed by the word “TaskID=”, then the task ID, date, and time separated by tabs in the usual manner.

CHAPTER 9

# Inventory Data Entry Methods

Submitting a new inventory record to 4th Quarter through 4D Open is not supported at this time

CHAPTER 10

# Advanced Topics

This chapter considers topics related to the handling of Batch Task records that will not needed in most cases.

## *Retrieving Batch Task Information*

### When a task is started

Insert, update and delete operations are handled by 4th Quarter's Batch Task system.

When a 4D Open client submits one of these tasks the 4D Open server performs some syntax and data checking, creates a Batch Task record, and returns to the client a related record ID and task ID number. The client does not need to wait for the actual process to be completed.

As soon as 4th Quarter creates a Batch Task record it attempts to process the task. The 4D Open client has already been informed of the Task ID number through the vBTErrorCode variable. The 4D Open client can locate the Batch Task record, or the record being updated or created by the task, using this Record ID or the Task ID number.

The vBTErrorCode variable will return the following information concatenated in the single text variable.

```
vBTErrorCode =  result value (integer) + Tab +
                {"TaskID=" + <Task ID> +Tab} +
                current date +Tab +
                current time
```

If the call has permanent errors, then the result value will be a negative integer value.

If the call is successful, then the result Value is the ID number of the highest level parent record relating to the data that's being inserted or updated. In the case of an invoice, the this will be the invoice ID number. In the case of a customer it will be the customer's ID number.

The task ID portion that follows only appears when a task record has been created to record the request. The task ID and the related record ID can be used to locate the Batch Task record, read values from it, or to delete it.

## Before a task is completed

4th Quarter may not be able to successfully process the batch task at its first try. It's also possible that 4th Quarter will be delayed in completing the task due to contention for records locked by other processes or other users.

If 4th Quarter cannot process the task upon its first attempt, it will pause for the number of ticks set by the administrator, and then attempt to process the job again.

This cycle will continue until the task is either successfully processed, or a maximum number of attempts has been reached. This maximum is set by the 4th Quarter administrator on the API page of the Maintenance screen.

After the 4D Open client retrieves the vBTErrorCode, the client can query 4th Quarter's Batch Task table for the task record with the corresponding ID number. The client can also query the 4th Quarter table in which the new or modified record is stored.

For example, if you've submitted a new customer record you can read the vBTErrorCode variable from the 4D Open server to determine the ID value of this record. If the value "1052" is returned, this means that the new customer will be assigned customer ID=1052. It also means that the Batch Task record in which the customer information is stored prior to being saved in the customer table has the value 1052 assigned to its "ref_RecID" field.

There may be other Batch Task records with the same value assigned to their ref_RecID fields. Only the combination of the table number and the record ID identifies a batch task record that acts on an unique record in the data base.

However, even this combination may not correspond to an unique Batch Task record. Other API calls may have been made to act upon this same record in the customer table, for example. If you need to locate the one task record that was created to handle a particular API call, then you must locate it by using its Task ID.

## Batch Status API

4th Quarter's "Batch_Status" API will retrieve the values of the fields of any Batch Task record (except the BLOB field) and place them in the `vy1BTText` text array.

4th Quarter can locate Batch Task record information if you supply it with the task ID. This ID value must be placed in the first element of a text array. This array is then copied into a BLOB and the BLOB is then passed to the 4D Open server along with the action parameter “BATCH_STATUS”.

Call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “BATCH_STATUS”
$3= BLOB (1-element text array whose first element contains the task ID.)

To locate a Batch Task record with task ID=1052 you would set up the BLOB as follows:

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($BatchTaskSpecifier;1)
$BatchTaskSpecifier{1}:="1052"
VARIABLE TO BLOB($BatchTaskSpecifier; $DataBlob)
```

**TABLE 45.** Batch Task Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 1 | X | Batch Task ID. | Used to find record with matching [Batch_Task] Task_ID value. |

Upon receiving this information the 4th Quarter API immediately searches for the specified [Batch_Task] record. It will assign a value to the vBTError-Code field if the request should fail for any of the follow reasons.

**TABLE 46.** Batch Task Information Error Values

| Error Code | Description |
|---|---|
| "" (blank) | Task has not yet completed the search. |
| "0" | [Batch_Task] record located. Values are in the vy1BTText array. |
| "-1" | Too few parameters submitted. |
| "-3" | Bad Task ID = <…>. |
| "-4" | Batch Task ID not supplied. |
| "-5" | [Batch_Task] record with task ID = <…> does not exist. |

**TABLE 46.** Batch Task Information Error Values

| Error Code | Description |
|---|---|
| "-6" | Changes to any 4Q tables are currently forbidden. |

## Reading Batch Task information

When the “Batch Status” tasks locates the indicated Batch Task record, it places the field values of this record in the vy1BTText array. The values in this array remain available to the 4D Open client until the client resets the value of the 4D Open server’s vBTErrorCode variable to a blank value.

Once the client erases the value vBTErrorCode variable, the server-side process and all of its variables, including the vy1BTText variable, disappear.

All Batch Task field values, except that stored in the BLOB field, are assigned to element of the vy1BTText array. The assignments are as follows.

**TABLE 47.** Batch Task field values

| Element | Value assigned to the vy1BTText Array on 4D Open server. |
|---|---|
| 1 | Batch Task ID |
| 2 | Value of ref_FileNum field. |
| 3 | Value of ref_RecID field. |
| 4 | Title given to task by the system. |
| 5 | Error code returned from last processing attempt. |
| 6 | Message the system assigned upon last processing attempt. |
| 7 | Attention value assigned by administrator. |
| 8 | Action value (originally submitted as $2). |
| 9 | Number of attempts made to process the task. |
| 10 | BLOB size in bytes. |
| 12 | Date task was submitted. |
| 13 | Time task was submitted. |
| 14 | Date task was successfully processed. |
| 15 | Time task was successfully processed. |
| 16 | Name of the 4D Open client who submitted the task. |

By referring to the error code in the 5-th element you can determine whether the task has been successfully processed or not. If this value is zero, then the task completed successfully. If it is negative, then the task has not yet been successfully completed.

## Global Lock-Outs

There are some maintenance procedures in 4th Quarter that require the database to be accessed by only one users. These operations include changing the fiscal year and running certain administration utilities. The are all procedures that would be run infrequently, if they're run at all.

Before running these procedures the system checks the number of users that are logged in to the database. However, it will not see any 4D Open connections.

When these database access limiting procedures are running they set a 4D semaphore whose name is stored in the interprocess variable <>vSYDB_LockoutFlag. The value stored in this variable can be read through a 4D Open call. When these procedures finish they clear this semaphore.

The _BT4QSubmitTask procedure checks for this semaphore before it begins any task that modifies data. These are all tasks that include the substrings "_Add", "_Modify", or "_Delete" in their batch task instruction parameter. If the <>vSYDB_LockoutFlag semaphore has been set, then the batch task will immediately exist and set the vBTErrorCode variable to "-6". Read-only calls can still be made.

Protect yourself from unexpectedly having your 4D Open requests rejected by testing for this semaphore yourself in the 4D Open client code before you send a task to the 4D Open server. Remember to clear this semaphore after testing it, as shown below, or you will lock users out of the database!

```
$4DOpenErrCode:=OP Get Process Variable (MyConnectID;
                "$MyProcessID"; "<>vSYDB_LockoutFlag";
                "v4QLockoutFlag")
$Err:=OP Set Semaphore (MyConnectID; v4QLockoutFlag;
                vSemaphoreState)
$4QIsLocked:=vSemaphoreState
$Err:=OP Clear Semaphore (MyConnectID; v4QLockoutFlag)
```

If, after these calls, the boolean variable $4QIsLocked is true, then you should not make any 4D Open calls that will update tables.

## Confirming the Success of a Batch Task

The 4D Open client may need to wait for the task to successfully complete. In some applications the client may even need to control the batch task record.

This situation might arise if the client needed to know additional information that will only become available after the task is complete. Or it could be that the client cannot proceed until they are sure the action was successfully.

## Batch Task Status

The client can determine the outcome of the batch process in either of two ways. Either by checking the batch task record directly, or by monitoring the vBTErrorText variable.

First note that if the vBTErrorCode variable returns a negative result code as a result of your initial call, then the request has been rejected and no batch task has been created. Only when you get a positive result code do you know that the task has been accepted and is either finished or in-process.

To determine the status of a task call the “Batch_Status” API, described above, during or after processing of the batch task.

If the value “-5” is returned in vBTErrorCode in response to the “Batch_Status” call, then you know the Batch Task record does not exist. This is either because:

- you have miss-specified its related table and record ID, or
- it has not yet been created, or
- it was created and either was or was not able to process successfully, and the Administrator subsequently deleted it, or
- it was created and processed successfully, was stored for the appropriate number of days, and then the system automatically overwrote it.

If any other negative value is returned, then you know the server could not understand or process your request. In this case the Batch Task record remains in the 4th Quarter system and may be resubmitted in the future.

Only when the value “0” is retrieved do you know the Batch Task information is available in the vy1BTText array. You can then check the value in the 5th element of the vy1BTText array. If this element has the value “0”, then you know the task has been successfully completed.

If you need to know more about the information stored in the newly inserted or updated records, then you can use 4D Open to search the record in the appropriate table.

## Stopping and Deleting a Batch Task

### To stop the task

If the actions of the 4D Open client are contingent upon the success of the submitted task, and if the task has not been successfully processed, then you can either wait until it is processed, or you can stop the task.

If you choose to wait, then you must continue to make periodic calls to the 4D Open server to check on the values of the Batch Task record, or periodically monitor the vBTErrorText variable.

If you choose to stop the Batch Task you can do this in either of two ways.

- Delete the Batch Task record using the "BATCH_DELETE" call. Do this if you want to be sure that the task won't be resubmitted by the Administrator in the future.

  This request is handled by the 4D Server through the creation of a separate 4D Open server process. This process may not be able to delete the indicated record if the same record remains loaded and locked in the process that originally created the record.

  For this reason, the process that originally created the batch task record should have terminated before you issue the command to delete the task record.
- Tell the submitting process on the server to halt any further processing attempts and to delete the Batch task record. This is done by assigning the value "BATCH_HALT" to the vOutsideControl variable on the 4D Open server. Remember that you must use the ***OP Set Process Variable*** to set this variable's value on the server.

  In response to this action the server will assign the value "-9" to the vBTErrorCode variable and the value "Batch_Deleted" to the vErrorText variable.

  Stopping a task that is currently being submitted can only be done when the submitting process is still running on the 4D Server.

  The action to stop this task, directed to this process, can only be done by the 4D Open client that submitted the task. Only that client has the process number that is necessary for setting the vOutsideControl variable using the ***OP Set Process Variable*** command.

In order to be certain that the Batch Task record has been deleted it would be wise to issue both of these command. This would handle the case where the server had already abandoned attempts to process the task between the time you last checked and the time you issued the instruction to halt the batch.

If you do set the control variable to stop the task and also submit a request to delete the task, then you should first assign the "Batch_Delete" value and sec-

ond submit the request to delete the task. If you perform these actions in the reverse order, then the deletion request may fail because the Batch Task may still be loaded and locked by the process that originally created it.

### To delete the task

To have 4th Quarter delete an existing [Batch_Task] record call the _BT4QSubmitTask method with the following parameters.

$1= Application, user, and method name concatenated in a text string.
$2= “BATCH_DELETE”
$3= BLOB (containing 1-element text array with task ID.)

For example, to delete a Batch Task record identified as task ID=1052 you would set up the BLOB as follows:

```
C_BLOB($DataBlob)
SET BLOB SIZE($DataBlob;0)
ARRAY TEXT($BatchTaskSpecifier;1)
$BatchTaskSpecifier{1}:="4568"
VARIABLE TO BLOB($BatchTaskSpecifier; $DataBlob)
```

**TABLE 48.** Batch Task Text Array Structure

| # | Req | Data | Description, type and formatting |
|---|---|---|---|
| 1 | X | Task ID. | Used to find record with matching [Batch_Task] Task_ID value. |

Upon receiving this information the 4th Quarter API immediately searches for the specified [Batch_Task] record. It will assign one of the following values to the vBTErrorCode field if the request fails.

**TABLE 49.** Batch Task Deletion Error Values

| Error Code | Description |
|---|---|
| "" (blank) | Task has not yet completed the search. |
| "0" | [Batch_Task] record located and deleted. |
| "-1" | Too few parameters submitted. |
| "-3" | Bad Task ID = <…>. |
| “-4” | Batch Task ID not supplied. |
| "-5" | [Batch_Task] Task_ID = <…> does not exist. |
| "-6" | Changes to any 4Q tables are currently forbidden. |
| “-7” | [Batch_Task] record is locked and cannot be deleted. |

## The _UTILTestBTSubmit Method

The API's are designed to be called from outside 4th Quarter through an inter-application protocol such as 4D Open. However, for the purposes of testing, they can also be called from within 4th Quarter using the _UTILTestBTSubmit method, designed especially for this purpose.

To call the _UTILTestBTSubmit you will need to write a method of your own, in the 4th Quarter application, that assigns values to variables and passes them as parameters to _UTILTestBTSubmit.

The _UTILTestBTSubmit method takes as parameters the value that are needed in calls to the API's. This method calls the indicated API in a separate process on the same machine. They are not called in a server process. An example of a call that creates an account record is as follows:

```
$vOffset:=0
$BTAction:="ACCOUNT_NEW/TEST_MODE"
$PrcsName:="API-Test"
$StackSize:=◊ vSYDfltPrcs
C_BLOB($TestBlob)
SET BLOB SIZE($TestBlob;0)
ARRAY TEXT(var1SYText;28)
var1SYText{1}:="541"  `Account ID
var1SYText{2}:="24"  `GL Account ID
var1SYText{3}:="-1"  `Use GL Defaults key value(1-1)
var1SYText{4}:=”456”  `Account Number
var1SYText{5}:=""  `Account Suffix
var1SYText{6}:=”Test Name”  `Account Name
var1SYText{7}:="-1"  `Equity Account ID
var1SYText{8}:="-1"  `Related File Number
var1SYText{9}:="-1"  `Related Record ID
var1SYText{10}:=""  `Related Record Name
var1SYText{11}:=""  `External Account Number
var1SYText{12}:=""  `Description
var1SYText{13}:="new test ref name"  `Reference Name
var1SYText{14}:=""  `Attention
var1SYText{15}:=""  `System Action
var1SYText{16}:="test addr text"  `Address Text
var1SYText{17}:="0"  `Use Related Record Address key value
                     (1,0)"
var1SYText{18}:="1"  `Active key value (1,-1)"
var1SYText{19}:="1"  `Access key value (1,-1)"
var1SYText{20}:="0"  `Finance %/Month
```

```
var1SYText{21}:="5"  `Days Grace
var1SYText{22}:="1"  `Monitor Balance (1,0)
var1SYText{23}:="500"  `Debit Caution
var1SYText{24}:="100"  `Debit Notify
var1SYText{25}:="-100"  `Credit Caution
var1SYText{26}:="-500"  `Credit Notify
var1SYText{27}:="2"  `Next Check Number
var1SYText{28}:="1"  `Autoallocation key (0,1)
VARIABLE TO BLOB(var1SYText;$TestBlob;$vOffset)

$Err:=_UTILTestBTSubmit ("_UTIL_4";"";$BTAction;
      $TestBlob;$PrcsName;$StackSize;->vErrorCode;
      ->vErrorText)
ARRAY TEXT(var1SYText;0)
```

Internal calls of this sort are not meant to fully duplicate what happens when calling the API's in a server process from a remote machine. While they do not perform exactly the same as a real 4D Open client connection, they do provide an easy means of testing the following:

- Packing and unpacking of the parameters,
- Timing and communication through the shared message variables.
- Access to messages and error codes in the API processes.

To use the *_UTILTestBTSubmit* method first assemble all the parameters in the same way you would if you were making a call from a remote application, such as from a 4D Open client. This includes packing text array information into a blob.

This information is then passed as parameters to the _UTILTestBTSubmit method. The _UTILTestBTSubmit method behaves in the same way as a 4D Open client application. That is:

- It calls the API's with the supplied parameters.
- It watches the value of the vBTErrorCode in the API process for an indication that the API has completed its task.
- It assigns a blank value to this variable in the API process. This causes the API process to come to an end.
- The _UTILTestBTSubmit returns the numeric value of the vBTErrorCode variable in $0.

The _UTILTestBTSubmit method also has built into it the ability to track the values assigned to the vBTErrorCode and vBTErrorText variables. It tracks when these values are assigned in the API, what values are assigned, when

the values are read, and what values are read in the _UTILTestBTSubmit process.

This audit trail is accumulated in the <>vBT1Text variable. This variable is written to a file on disk before the _UTILTestBTSubmit finishes. To engage this auditing feature you must append the string “Test_Mode” to the action parameter $3 that is passed to the API.

You can also call _UTILTestBTSubmit without providing sufficient information for the indicated API call of your choice. The _UTILTestBTSubmit will supply blank parameters of the correct type in the API calls and you will be able to monitor the API’s error handling response.

_UTILTestBTSubmit takes the following parameters:

$1= Name of the calling method.
$2= Action code read by _UTILTestBTSubmit.
$3= Action code to pass to API (optional).
Append the string “Test_Mode” to $3 in order to activate the auditing of the communication variables.
$4= Blob to pass to API (optional).
$5= Name to assign to spawned API process (optional).
$6= Stack size for the new process in bytes (optional).
$7= Pointer to error code text variable (optional).
$8= Pointer to error description text variable (optional).
$0= Longint error code returned from API.

If the optional parameters are supplied, then they must be of the correct type. If they are not supplied, then the _UTILTestBTSubmit method will provide default values to the API where necessary.

_UTILTestBTSubmit will supply most parameters with no data. If the API expects these optional parameters to have data in then, then the blank values will trigger an error handling response by the API.

If all the correct data is provided, then the API will proceed in exactly the same manner as if it had been called from a remote client, except that it will not be running as a server process. New records will be created, existing records modified, and deleted.

- Using this test method makes it easier for you to trace the API.
- You can step through the API’s execution.
- Alert messages encountered during the API’s execution will be displayed on screen. Had the API executed on the server these message would not appear, though they would still be logged as records in the LogHistory table.

- You can generate an audit history of the vBTErrorCode and vBTErrorText variable.

## Bug Reporting, Problem Testing

The 4D Open architecture makes it more difficult to identify the source of problems, and more difficult for Braided Matrix to locate bugs. As a rule, we cannot reproduce the situation that occurs at your site as accurately as when everything is occuring within the context of the 4th Quarter program.

Braided Matrix can only fix bugs that occur on the server side of the 4D Open equation. If you are having trouble establishing or maintaining a connection between the 4D Open client and 4th Quarter, acting as the 4D Open server, then the problem is probably related to the network or the 4D Open configuation. Such problems are beyond our control.

Before coming to the creators of 4th Quarter for help you must first learn to use 4D Open correctly from the the client side, and be fairly certain that 4D Open communications are working. Once this level of integrity has been established, there are still many issues arising from the reaction of the 4D Open server that might confuse of confound you.

If you are having problems, the first thing to do is study the structure of the message returned in the vBTErrorCode variable. This is the source of nearly all the information that is available to you from 4Q, and several pieces of information are concatenated into this variable. The vBTErrorText variable provide a verbal explanation of the error codes, but this may n ot be complete. Indeed, the description of conditions given here might be confusing or even wrong. Compare this message with the values in the vBTErrorCode variable and the list of error codes in previous chapters.

The next thing to do is to check the Batch Task table to see if the calls that are being made to 4Q are being logged there. Only calls that make it past the structural integrity checks are candidated for being saved as records in the batch task table, so the absence of any records there does not necessarily indicate a failure on the 4D Open server side.

If you need to communicate with Braided Matrix, then chances are that we will need to try to reproduce your situation before we can resolve your problem. By far the easiest way for us to help you is if you provide us with these three things:

**Data File:** We need your data file to follow how 4Q is reponding to the information that you provide in your 4D Open call. Data files compact over 80% so it is often possible to email us a large data file after it has been compressed. If

it does not compress enough, then you may have to put it on a CD and send it through the mail.

**Command Syntax:** We need the exact command parameters that you are sending to 4Q through the Execute On Server command. You can easily provide this in a few words.

**Parameter Data:** We need the BLOB that you are sending to 4Q, if a BLOB is being used. You can easily provide us with this by writing the BLOB to a file on disk, and then emailing us the file. The BLOB should be written to disk using 4D's Send Variable command. You can then put the BLOB in the same compressed archive as contains your data file.

If you can provide us with these things, then we can reproduce the exact situation seen from within 4Q once the 4D Open connection has been established and the 4Q API has passed the data in to 4th Quarter.