Getting Started with the TWS C++ API Guide

Interactive Brokers
The Professional's Gateway to the World's Markets

Getting Started with the TWS C++ API
March 2011
Supports TWS API Release 9.64

Any symbols displayed within these pages are for illustrative purposes only, and are not intended to portray any recommendation.

Contents

1

Introduction

You might be looking at this book for any number of reasons, including:

- You love IB's TWS, and are interested in seeing how using its API can enhance your trading.
- You use another online trading application that doesn't provide the functionality of TWS, and you want to find out more about TWS and its API capabilities.
- You never suspected that there was a link between the worlds of trading/financial management and computer programming, and the hint of that possibility has piqued your interest.

Or more likely you have a reason of your own. Regardless of your original motivation, you now hold in your hands a unique and potentially priceless tome of information. Well, maybe that's a tiny bit of an exaggeration. However, the information in this book, which will teach you how to access and manage the robust functionality of IB's Trader Workstation through our TWS C++ API, could open up a whole new world of possibilities and completely change the way you manage your trading environment. Keep reading to find out how easy it can be to build your own customized trading application.

If you are a Financial Advisor who trades for and allocates shares among multiple client accounts and would like more information about using the ActiveX API, see the Getting Started with the TWS C++ API for Advisors Guide.

How to Use this Book

Before you get started, you should read this section to learn how this book is organized, and see which graphical conventions are used throughout.

Our main goal is to give active traders and investors the tools they need to successfully implement a custom trading application (i.e. a trading system that you can customize to meet your specific needs), and that doesn't have to be monitored every second of the day. If you're not a trader or investor you probably won't have much use for this book, but please, feel free to read on anyway!

We should also tell you that throughout this book we use the TWS C++ API sample application to demonstrate how we implemented the API. However, our sample application is not our primary focus. Our main objective is to introduce you to the methods and parameters in the C++ API that you will need to learn to build your own custom trading application. You can use the sample application as a starting point.

Throughout this book, we use the acronym "TWS" in place of "Trader Workstation." So when you see "TWS" anywhere, you'll know we're talking about Trader Workstation.

Before you read any further, we need to tell you that this book focuses on the TWS side of the C++ API - we don't really help you to learn C++. If you aren't a fairly proficient C++ programmer, or at least a very confident and bold beginner, this may be more than you want to take on. We suggest you start with a beginner's C++ programming book, and come back to us when you're comfortable with the language.

Organization

We've divided this book into five major sections, each of which comprises a number of smaller subsections, and each of those have even smaller groupings of paragraphs and figures…well, you get the picture. Here's how we've broken things down:

Part 1: Introducing the TWS C++ API

The chapters in this section help you answer those important questions you need to ask before you can proceed - questions such as "What can TWS do for me?" and "Why would I use an API?" and "If I WERE to use an API, what does the C++ API have to offer me?" and even "What other API choices do I have?"

If you already know you want to learn about the TWS API, just skip on ahead.

Part 2: Preparing to Use the TWS C++ API

Part 2 walks you through the different things you'll need to do before your API application can effectively communicate with TWS. We'll help you download and install the API software, configure TWS, and get the sample application up and running. A lot of this information is very important when you first get started, but once it's done, well, it's done, and you most likely won't need much from this section once you've completed it.

Part 3: Market Data

Part 3 gets you working with the C++ sample application to get market data. You'll learn how to request, receive and cancel market data, market depth, historical data, real time bars, run market scanners and get contract data. We'll tell you exactly what methods you need to use to send info to TWS, and just what TWS will send you back. We've already documented the method parameters, descriptions and valid values in the *API Reference Guide*, but we have provided a lot of those details here for your convenience.

Part 4: Orders and Executions

Part 4 takes your through the order-related tasks in the C++ API sample application. You'll learn how the API handles the process of placing and canceling an order, viewing open orders, and viewing executions. Here too we provide the methods, events and parameters used for these trading tasks.

Part 5: Additional Tasks

Part 5 continues your path through the C++ API sample application by describing the methods, events and parameters used for the rest of the buttons in the sample application, including how to get the current system time and how to request and cancel news subscriptions.

Part 6: Where to Go from Here

After filling your head with boatfuls of API knowledge, we wouldn't dream of sending you off empty-handed! Part 6 includes some additional information about linking to TWS using our C++ API, then tells you how to keep abreast of new API releases (which of course means new features you can incorporate into your trading plan), how to navigate the Interactive Brokers website to find support and information, and what resources we recommend to help you answer questions outside the realm of IB support, questions such as "Why isn't my Visual Studio working?"

Footnotes and References

[1]Any symbols displayed are for illustrative purposes only and are not intended to portray a recommendation.

Icons

TWS-Related

When you see this guy, you know that there is something that relates specifically to TWS: a new feature to watch for, or maybe something you're familiar with in TWS and are looking for in the API.

C++ Tip

These C++tips are things we noted and think you might find useful. They don't necessarily relate only to TWS. We don't include too many of these, but when you see it you should check it out - it will probably save you some time.

Important!

This shows you where there is a particularly useful or important point being made.

Take a Peek!

You may want to take a peek, but it isn't the end of the world if you don't.

Go Outside!

This icon denotes references outside of this book that we think may help you with the current topic, including links to the internet or IB site, or a book title.

Document Conventions

Here's a list of document conventions used in the text throughout this book.

Convention	Description	Examples
Bold	Indicates: • menus • screens • windows • dialogs • buttons • tabs • keys you press • names of classes and methods	When you click the Req Mkt Data button… Press Ctrl+C to copy…

Italics	Indicates: • commands in a menu • objects on the screen, such as text fields, check boxes, and drop-down lists	To access the users' guide, under the Software menu, select *Trader Workstation*, then click *Users' Guide*.
`Code samples`	Code samples appear gray boxes throughout the book.	See below.

```
m_pClient->reqMktData( m_dlgOrder->m_
        m_dlgOrder->m_genericTicks, m_
```

2

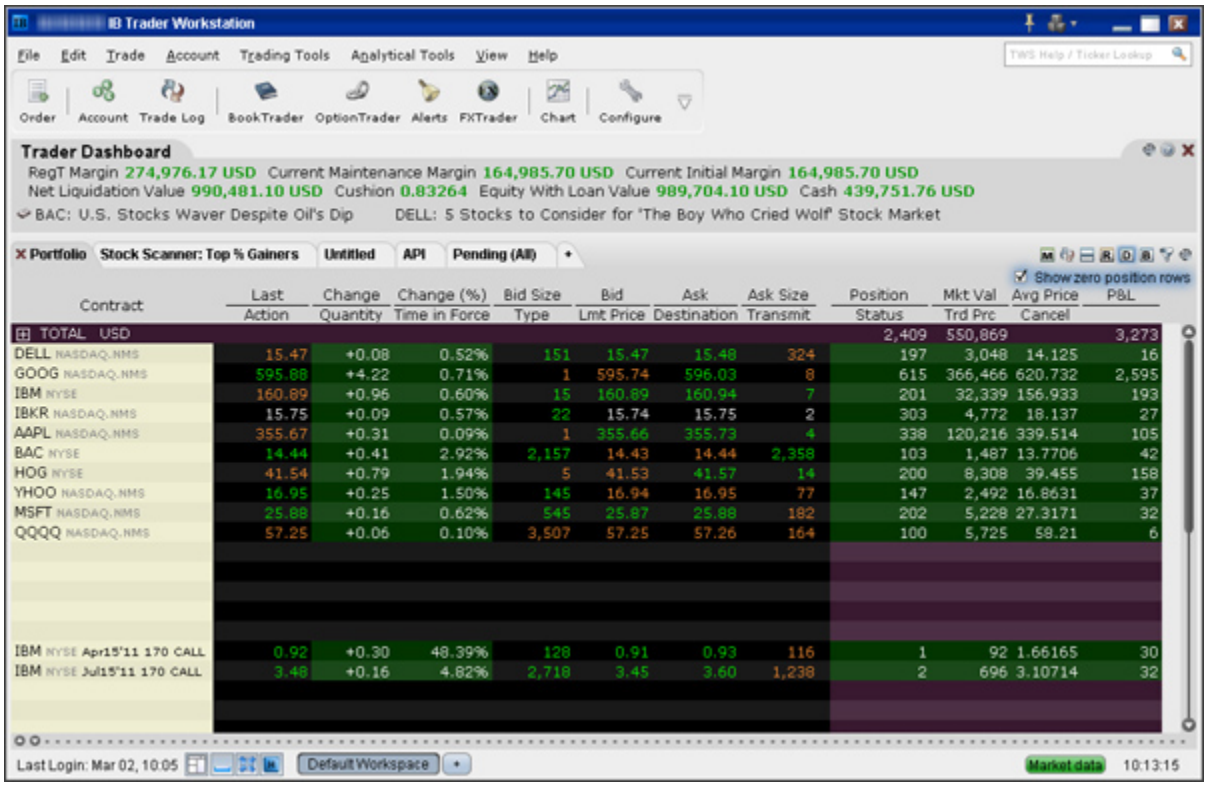
TWS and the C++ API

The best place to start is by getting an idea of what Trader Workstation (TWS), is all about. In this section, first we'll describe TWS and some of its major features. Then we'll explain how the API can be used to enhance and customize your trading environment. Finally, we'll give you a summary of some of the things the C++ API can do for you!

Here's what you'll find in this section:

Chapter 1 - What is Trader Workstation?

Interactive Brokers' Trader Workstation, or TWS, is an online trading platform that lets you trade and manage orders for all types of financial products (including stocks, bonds, options, futures and Forex) on markets all over the world - all from a single spreadsheet-like screen.



To get a little bit of a feel for TWS, go to the IB website and try TWS demo application. Its functionality is slightly limited and it only supports a small number of symbols, but you'll definitely get the idea. Once you have an approved, funded account you'll also be able to use PaperTrader, our simulated trading tool, with paper-money funding in the amount of $100,000, which you can replenish at any time through TWS Account Management.

What Can You Do with TWS?

So, what can you do with TWS? For starters, you can:

- Send and manage orders for all sorts of products (all from the same screen!);
- Monitor the market through Level II, NYSE Deep Book and IB's Market Depth;
- Keep a close eye on all aspects of your account and executions;
- Use Technical, Fundamental and Price/Risk analytics tools to spot trends and analyze market movement;
- Completely customize your trading environment through your choice of modules, features, tools, fonts and colors, and user-designed workspaces.

Basically, almost anything you can think of TWS can do - or will be able to do soon. We are continually adding new features, and use the latest technology to make things faster, easier and more efficient. As a matter of fact, it was this faith in technology's ability to improve a trader's success in the markets (held by IB's founder and CEO Thomas Peterffy) that launched this successful endeavor in the first place. Since the introduction of TWS in 1995, IB has nurtured this relationship between technology and trading almost to the point of obsession!

A Quick Look at TWS

This section gives you a brief overview of the most important parts of TWS.

The TWS Quote Monitor

First is the basic TWS Quote Monitor. It's laid out like a spreadsheet with rows and columns. To add tickers to a page, you just click in the Underlying column, type in an underlying symbol and press Enter, and walk through the steps to select a product type and define the contract. Voila! You now have a live market data line on your trading window. It might be for a stock, option, futures or bond contract. You can add as many of these as you want, and you can create another window, or trading page, and put some more on that page. You can have any and all product types on a single page, maybe sorted by exchange, or you can have a page for stocks, a page for options, etc. Once you get some market data lines on a trading page, you're ready to send an order.

The Order Ticket

What? An order ticket? Sure, we have an order ticket if that's what you really want. But we thought you might find it easier to simply click on the bid or ask price and have us create a complete order line instantly, right in front of your eyes! Look it over, and if it's what you want click a button to transmit the order. You can easily change any of the order parameters right on the order line. Then just click the green Transmit guy to transmit your order! It's fast and it's easy, and you can even customize this minimal two-click procedure (by creating hotkeys and setting order defaults for example) so that you're creating and transmitting orders with just ONE click of the mouse.

Real-Time Account Monitoring

TWS also provides a host of real-time account and execution reporting tools. You can go to the Account Window at any time to see your account balance, total available funds, net liquidation and equity with loan value and more. You can also monitor this data directly from your trading window using the Trader Dashboard, a monitoring tool you can configure to display the last price for any contracts and account-related information directly on your trading window.

So - TWS is an all-inclusive, awesome powerful trading tool. You may be wondering, "Where does an API fit in with this?" Read on to discover the answer to that question.

For more information on TWS, see the TWS Users' Guide on our web site.

Chapter 2 - Why Use the TWS C++ API?

OK! Now that you are familiar with TWS and what it can do, we can move on to the amazing API. If you actually read the last chapter, you might be thinking to yourself "Why would I want to use an API when TWS seems to do everything." Or you could be thinking "Hmmmm, I wonder if TWS can… fill in the blank?" OK, if you're asking the first question, I'll explain why you might need the API, and if you're asking the second, it's actually the API that can fill in the blank.

TWS has the capability to do tons of different things, but it does them in a certain way and displays results in a certain way. It's likely that our development team, as fantastic as they are, hasn't yet exhausted the number of features and way of implementing them that all of you collectively can devise. So it's very likely that you, with your unique way of thinking, will be or have been inspired by the power of TWS to say something like "Holy moly, I can't believe I can really do all of this with TWS! Now if I could only just (fill in the blank),my life would be complete!"

That's where the API comes in. Now, you can fill in the blank! It's going to take a little work to get there, but once you see how cool it is to be able to access functionality from one application to another, you'll be hooked.

TWS and the API

In addition to allowing you pretty much free reign to create new things and piece together existing things in new ways, the API is also a great way to automate your tasks. You use the API to harness the power behind TWS - in different ways.

Here's an analogy that might help you understand the relationship between TWS and the API. Start by imagining TWS as a book (since TWS is constantly being enhanced, our analogy imagines a static snapshot of TWS at a specific point in time). It's the reference book you were looking for, filled with interesting and useful information, a book with a beginning, middle and end, which follows a certain train of logic. You could skip certain chapters, read Chapter 10 first and Chapter 2 last, but it's still a book. Now imagine, in comparison, that the API is the word processing program in which the book was created with the text of the book right there. This allows you access to everything in the book, and most importantly, it lets you continually change and update material, and automate any tasks that you'd have to perform manually using just a book, like finding an index reference or going to a specific page from the table of contents.

The API works in conjunction with TWS and with the processing functions that run behind TWS, including IB's SmartRouting, high-speed order transmission and execution, support for over 40 orders types, etc. TWS accesses this functionality in a certain way, and you can design your API to take advantage of it in other ways.

Available API Technologies

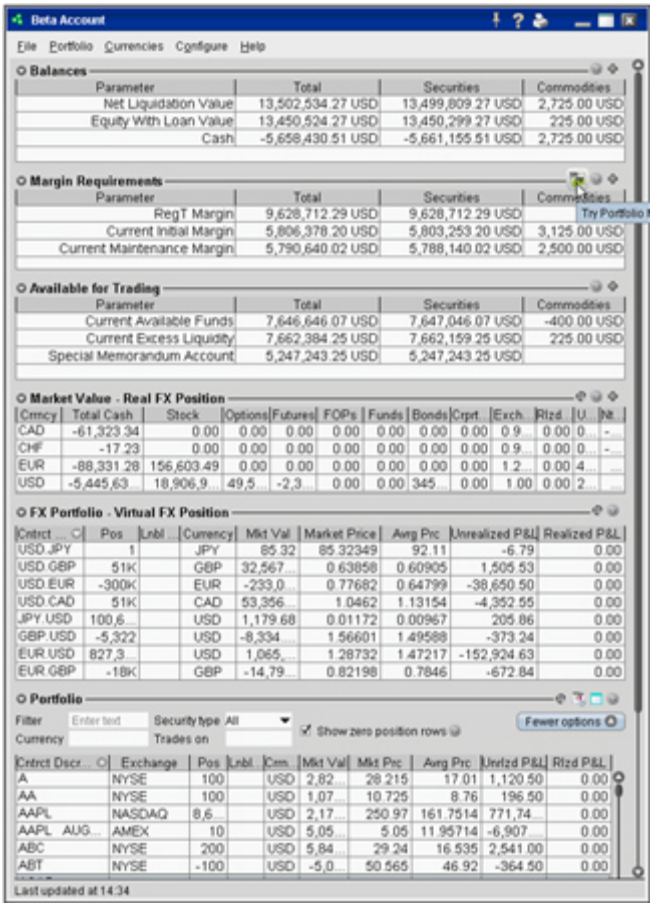
IB provides a suite of custom APIs in multiple programming languages, all to the same end. These include Java, C++, Active X for Visual Basic, and DDE for Excel (Visual Basic for Applications, of VBA). This book focuses specifically on just one, the C++ version. Why would you use C++ over the other API technologies? The main reason might be that you are a Visual Basic expert. If you don't know any other programming language, you should take a look at the DDE for Excel API, which has a smaller learning curve. But if you know Visual Basic, this platform offers more flexibility than the DDE for Excel API and provides reasonably high performance. The C++ API is supported only on Windows, however.

For more information about our APIs, see the Application Programming Interfaces page on our web site.

An Example

It's always easier to understand something when you have a real life example to contemplate. What follows is a simple situation in which the API could be used to create a custom result.

TWS provides an optional field that shows you your position-specific P&L for the day as either a percentage or an absolute value. Suppose you want to modify your position based on your P&L value? At this writing, the only way to do this would be to watch the market data line to see if the P&L changed, and then manually create and transmit an order, but only if you happened to catch the value at the right point. Hmmmmm, I don't think so! Now, enter the API! You can instruct the API to automatically trigger an order with specific parameters (such as limit price and quantity) when the P&L hits a certain point. Now that's power! Another nice benefit of the API is that it gives you the ability to use the data in TWS in different ways. We know that TWS provides an extensive Account Information window that's chock-full of everything you'll ever want to know about your account status. The thing is, it's only displayed in a TWS window, like the one on the next page.



Lovely though it is, what if you wanted to do something else with this information? What if you want it reflected in some kind of banking spreadsheet where you log information for all accounts that you own, including your checking account, Interactive Brokers' account, 401K, ROIs, etc? Again - enter the API!

You can instruct the API to get any specific account information and put it wherever it belongs in a spreadsheet. The information is linked to TWS, so it's easy to keep the information updated by simply linking to a running version of TWS. With a little experimenting, and some help from the *API Reference Guide* and the *TWS Users' Guide*, you'll be slinging data like a short-order API chef in no time!

There are a few other things you must do before you can work with the TWS C++ API. The next chapter gets you geared up and ready to go.

3

Preparing to Use the C++ API

Although the API provides great flexibility in implementing your automated trading ideas, all of its functionality runs through TWS. This means that you must have a TWS account with IB, and that you must have your TWS running in order for the API to work. This section takes you through the minor prep work you will need to complete, step by step.

Here's what you'll find in this section:

- Chapter 3 - Install an IDE
- Chapter 4 - Download the API Software
- Chapter 5 - Connect to the C++ API Sample Application

We want to tell you again that this book focuses on the TWS side of the C++ API - we don't really help you to learn C++. Unless you are a fairly proficient C++ programmer, or at least a very confident and bold beginner, this may be more than you want to take on. We suggest you start with a beginner's C++ programming book, and come back to us when you're comfortable with the language.

Chapter 3 - Install an IDE

OK, well we've already said that you need to know C++ before you can successfully implement your own TWS C++ API application, and there's a good chance you already have the tools you'll need downloaded and installed. But in case you don't, we'll quickly walk you through what you need, which is simply an integrated development environment (IDE) that supports Microsoft C++, as well as Microsoft Visual Basic and the Microsoft .NET framework.

In this book we use Microsoft Visual Studio 2008 as the IDE of choice. We'll try to keep the Visual Studio-specific instructions to a minimum, but if you're using another IDE you'll have to interpret those instructions to fit your C++ development environment. If you're using Visual Studio 2008 and aren't totally familiar with it, we recommend browsing through the How Do I section of the online help, which you can access from Visual Studio's Help menu.

Once you have your C++ development environment installed, you can go to the IB website and download the TWS API software.

Chapter 4 - Download the API Software

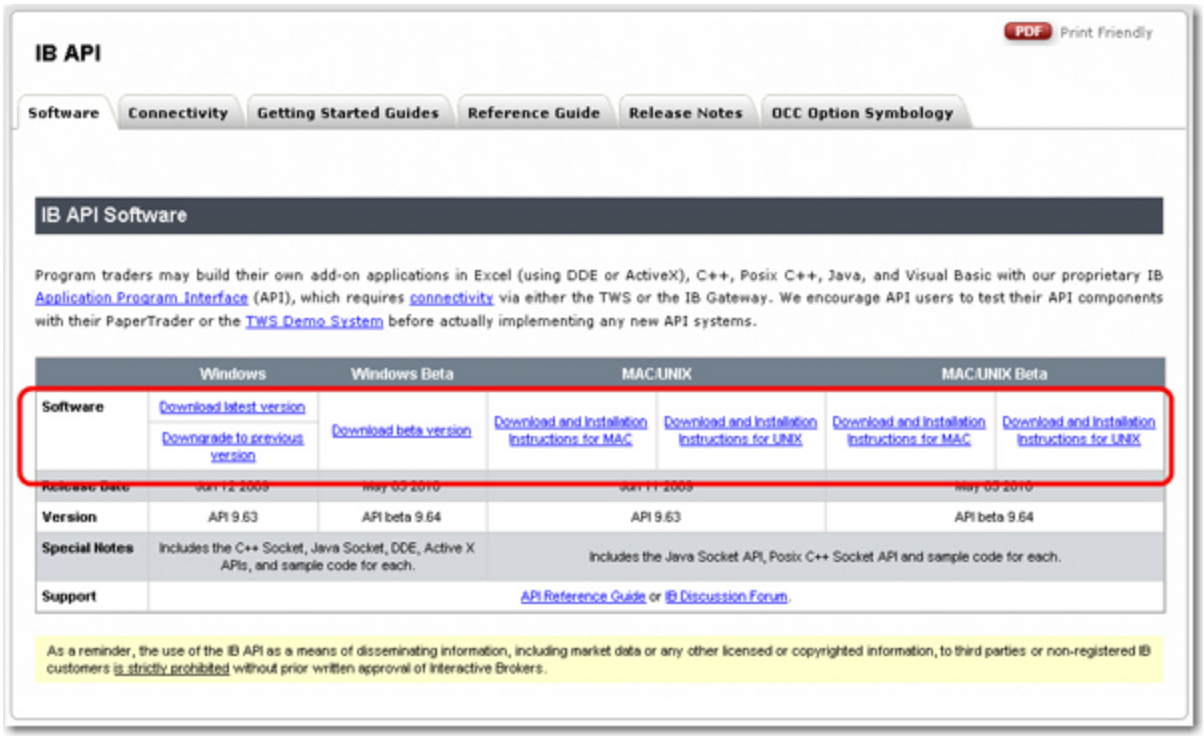
Next, you need to download the API software from the IB website.

Step 1: Download the API software.

This step takes you out to the IB website at http://individuals.interactivebrokers.com/en/p.php?f=programInterface&p=a&ib_entity=lic. The menus are along the top of the homepage. Hold your mouse pointer over the Trading menu, then click *API Solutions*.

On the API Solutions page, click the IB API button on the left side of the page.

This displays the IB API page which shows a table with links to software downloads that are compatible with Windows, MAC or Unix platforms. When available, there will also be a Windows Beta version of the software. Look across the top of the table and find the OS you need.



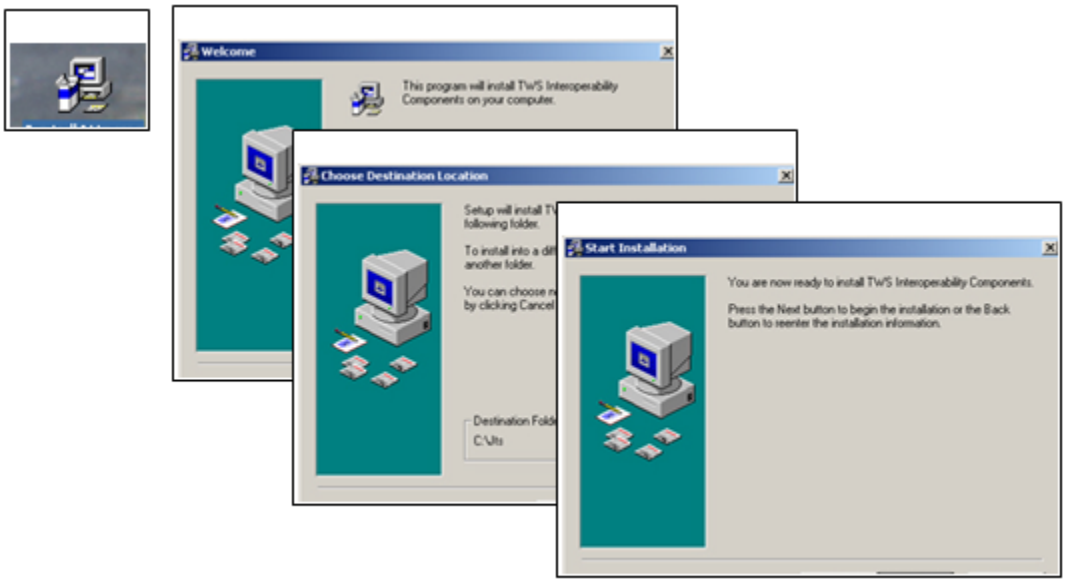
In the Windows column, click *Download Latest Version*. This opens a File Download box, where you can decide whether to save the installation file, or open it. We recommend you choose Save and then select a place where you can easily find it, like your desktop (you choose the path in the *Save in* field at the top of the Save As box that opens up). Once you've selected a good place to put it, click the Save button. It takes seconds to download the executable file. Note that the API installation file is named for the API version; for example, InstallAX_960.

We'll usually be stressing just the opposite, but at this point, you need to make sure TWS is not running. If it is, you won't be able to install the API software.

Step 2: Install the API software.

Next, go to the place where you saved the file (for example, your desktop or some other location on your computer), and double-click the API software installation file icon. This starts the installation wizard, a simple process that displays a series of dialogs with questions that you must answer.

Remember where the installation wizard installs the application. You'll need this information later when you open the API application in Excel.



Once you have completed the installation wizard, the sample application installs, and you're ready to open the C++ sample application, connect to TWS, and get started using the sample application!

Chapter 5 - Connect to the C++ Sample Application

OK, you've got all the pieces in place. Now that we're done with the prep work, it's time to get down to the fun stuff.

Although the API provides great flexibility in implementing your automated trading ideas, all of its functionality runs through TWS. This means that you must have a TWS account with IB, and you must have TWS running in order for the API to work. This section describes how to enable TWS to connect to the C++ API. Note that if you don't have an account with IB, you can use the Demo TWS system to check things out.. If you DO have an account, we recommend opening a linked PaperTrader test account, which simulates the TWS trading environment, and gives you $100,000 in phantom cash to play with.

Enabling TWS to support the API is probably the simplest step you'll encounter in this book. It's probably more difficult to actually remember to log into TWS before you run the API!

Running the C++ API Sample Application

Here's how to connect to the Visual Basic sample application:

Step 1: Log into TWS.

OK, log into TWS, or run the Demo available on the Demo tab of the Trader Workstation page on our website.

Step 2: Enable TWS to support the C++ API.

Now look up at the top of the trading window, and you'll see the menu bar. Click the Edit menu, and then click *Global Configuration*. In the Configuration window, click *API* in the left pane, then click *Settings*, which reveals several options on the right side of the window. Check the *Enable ActiveX and Socket Clients* check box and click OK.

Step 3: Run the C++ Sample Application.

We've included a complete C++ client with our API software. To run this sample application, go to your TWS API installation folder (typically *C:\IB_API_X_XX*, where *X_XX* is the API version number), then open the *TestSocketClient\Release* folder and run the file named *client2.exe*.

Running the C++ API Sample Application from Visual Studio 2008

If you prefer, you can run the VB.NET sample application from within Microsoft Visual Studio 2008. Here's how:

Step 1: Log into TWS.

This step is the same for the VB.NET sample as it was for the Visual Basic sample; we're including it here so you don't have to turn the page.

OK, log into TWS, or run the Demo available on the Demo tab of the Trader Workstation page on our website.

Step 2: Enable TWS to support the C++ API.

This step is the also the same for both versions of the sample application; again, we're repeating it for your convenience.

Click the Edit menu, and then click *Global Configuration*. In the Configuration window, click *API* in the left pane, then click *Settings*, which reveals several options on the right side of the window. Check the *Enable ActiveX and Socket Clients* check box and click OK.

Step 3: Run the C++ API Sample Application.

These instructions assume that you are using Microsoft Visual Studio 2008. Our C++ API sample application code was developed as a C++6 project. If you are using a different C++ development environment, adjust the instructions below accordingly.

Here's how to do this:

1 Download and install the latest version of the API software.

2 Create the folder where all project related files will be located. For example, *C:\SampleSocketClient*.

3 Copy the folders *TestSocketClient*, *Shared* and *SocketClient* into the folder you created in Step 2.

4 In Visual Studio, select New > Project From Existing Code from the File menu.

5 Select Visual C++ as the project type.

6 In the Create New Project from Existing Code Files dialog:

   a For the Project File Location, select the folder you created in Step 2.

   b For the Project Name, type SampleSocketClient.

   c Click Finish.

7 Right-click the project in the Solution Explorer and select Properties. In the Property Pages dialog:

   a On the left side of the dialog, select Configuration Parameters > C/C++> General. In the Additional Include Directories field on the right side of the dialog, type:

      ./Shared;./SocketClient/src

   b On the left side of the dialog, select Configuration Parameters > C/C++ > Code Generation. In the Runtime Library field on the right side of the dialog, select Multi-threaded (/MT).

8 Click OK to save your changes.

9 Build the project.

In case of errors:

If the C++ sample application won't build or run due to errors, try the following steps:

1 In the Visual Studio 2008 Solution Explorer, remove the following files from the project (select the item from the list and press Delete):

- Remove *DlgShareAllocation.h* from the list of Header Files.
- Remove *DlgShareAllocation.cpp* from the list of Source Files.
- Remove *TwsSocketClient.lib* from the project.

2 In the Solution Explorer, add the following existing items from the *TestSocketClient* folder in your TWS API installation folder (right-click the Header Files/Source Files folder, then select Add > Existing Item):

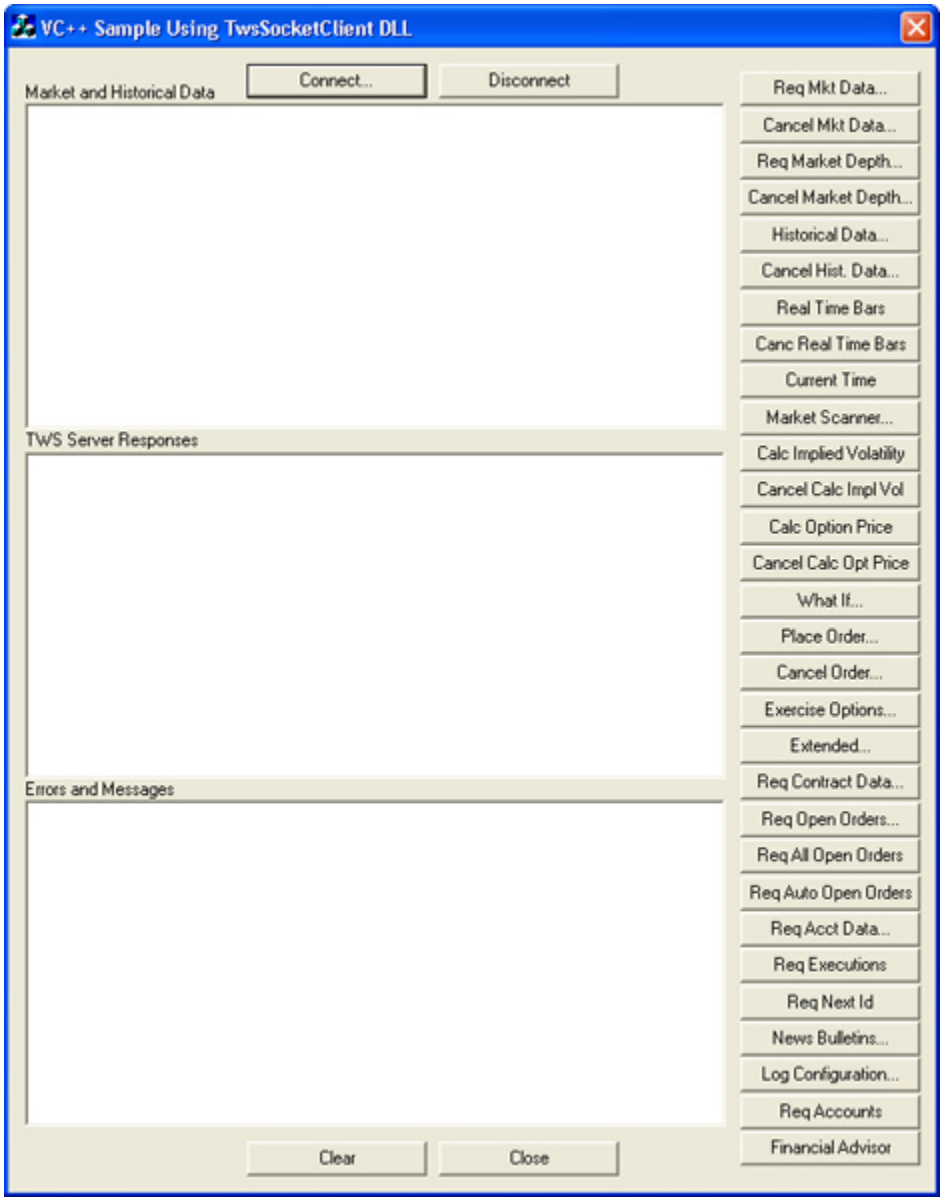
- Add *DlgUnderComp.h* to the list of Header Files.
- Add *DlgUnderComp.cpp* to the list of Source Files.

If you are running TWS API Version 9.6 or higher, you must also add DlgAlgoParams.h to the list of Header Files and DlgAlgoParams.cpp to the list of Source Files in the project.

3 In the Solution Explorer, add the following existing items from the *SocketClient/src* folder in your TWS API installation folder to the list of Source Files (right-click the Source Files folder, then select Add > Existing Item):

- *EClientSocket.cpp*
- *MySocket.cpp*

4 Press F5 to run the sample client, which is pictured on the next page.



What's Next

You're now ready to start looking at how our C++ sample application uses our TWS C++ API. Part 3 focuses on market data-related trading tasks defined by the action buttons in the sample application. We'll take a quick, general look at what's going on behind the GUI, then we'll walk through the basics of requesting market data using the TWS C++ API.

4

Market Data

You've completed the prep work, and you have the C++ sample application up and running. This section of the book starts with a description of the basic framework of the sample application, then reviews the TWS C++ API methods associated with each trading task.

This section describes how to connect the sample application to TWS and how to perform market data-related tasks such as requesting and canceling market data, historical data and real time bars, as well as how to subscribe to market scanners and get contract data. We'll show you the methods, events and parameters behind these trading tasks.

Here's what you'll find in this section:

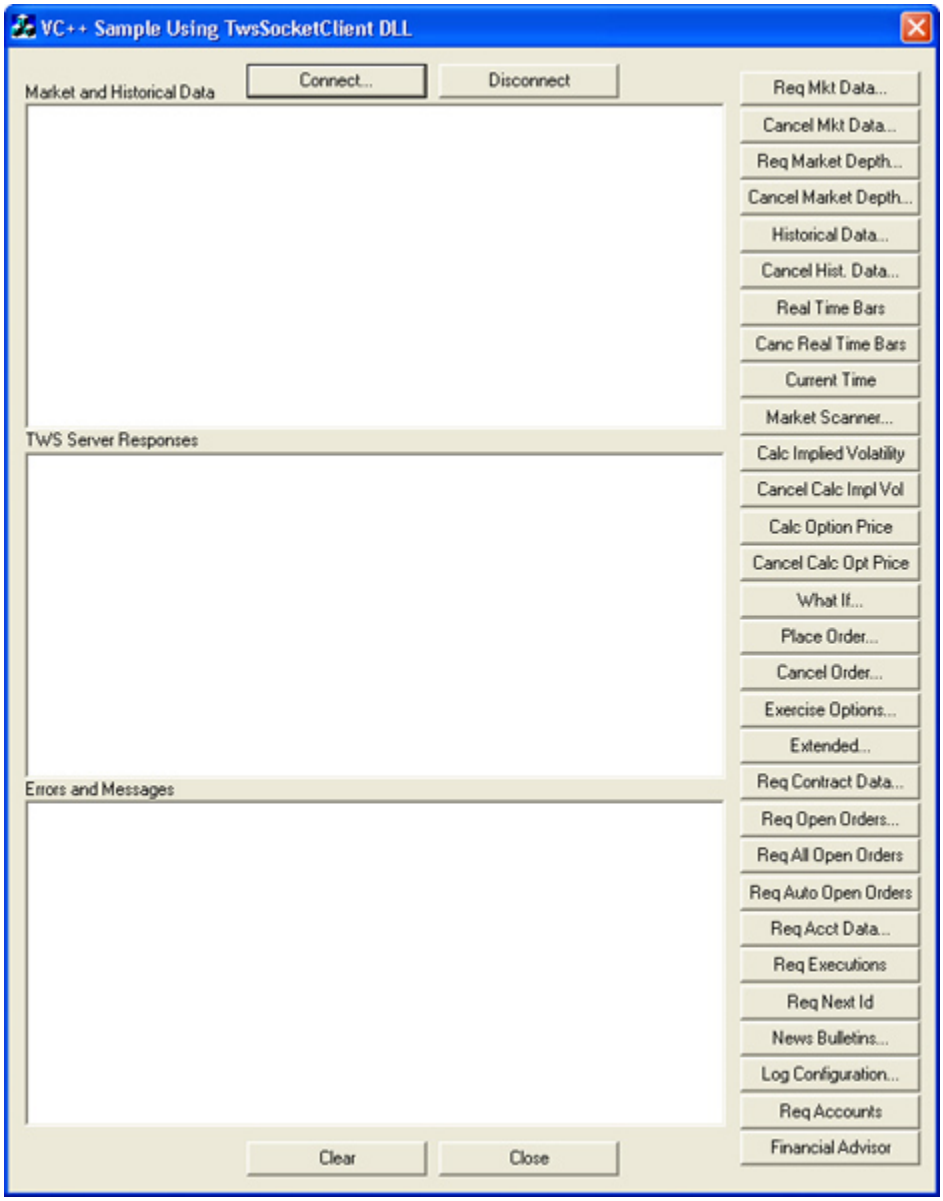
Using the C++ sample application is a good way to practice locating and using the reference information in the API Reference Guide. With the sample program, you can compare the data in the sample message with the method parameters in the *API Reference Guide.*

Chapter 6 - Connecting to TWS

This chapter describes the basic framework of the C++ sample application and what happens when you connect and disconnect to a running instance of TWS.

C++ Sample Application

Let's take a look at the C++ sample application and the C++ API. Here's the C++ sample application when you first run it:



Source Code

Our C++ API code includes many source files, including the source code for our sample application and the source code for our C++ API. For the purposes of this guide, you should pay attention to these files in particular:

- client2Dlg.cpp - The main source file that includes all of the EClientSocket methods, which send messages to TWS, the EWrapper functions, which return data from TWS, and the SocketClient properties, which contain groups of properties for things like contracts and orders.
- Dlg*.cpp - The source files for the many dialog boxes used in our C++ sample application. For example, the Connect dialog code is stored in DlgConnect.cpp.

These files are NOT part of our C++ API code but are part of the C++ sample application code. Throughout this guide, however, we will use code samples primarily from *client2Dlg.cpp*, as well as some code from the individual Dlg*.cpp files.

Our C++ API code contains code that doesn't directly apply to the purpose and content of this guide. If you are an experienced C++ programmer, you can look through all of our source code, including the code used for the sample application itself, but this is not necessary to learn the methods, functions and parameters in our C++ API.

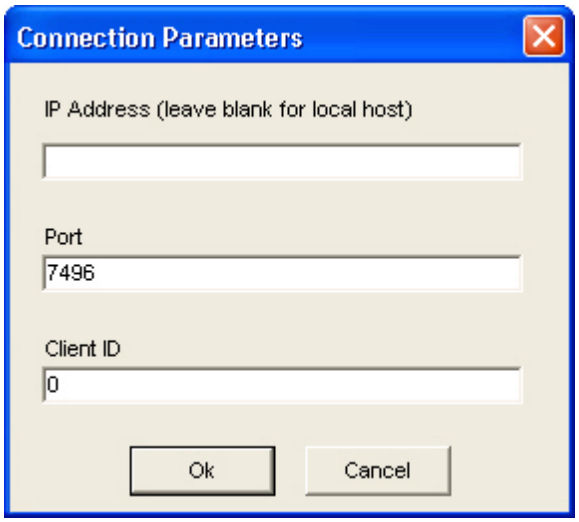
A Look at *client2Dlg.cpp*

If you open the file *client2Dlg.cpp* in your IDE, you will see a lot of C++ code. While using this guide, a few items in the file are of particular interest:

- #includes - The include statements include all of the header files needed by this source file.
- MESSAGE MAP - This section maps all of the buttons in the sample application graphical user interface (GUI) to related "On" methods.
- "On" methods - Methods included in *client2Dlg.cpp* that call the SocketClient methods in our C++ API. When you click a button in our sample application, an On method calls the related C++ API SocketClient method. For example, when you click the Connect button in the sample appliction to connect to TWS, OnConnect() calls our eConnect() SocketClient method. The On methods are not part of our C++ API code but are part of the code needed by our sample application code.

What Happens When I Click the Connect Button?

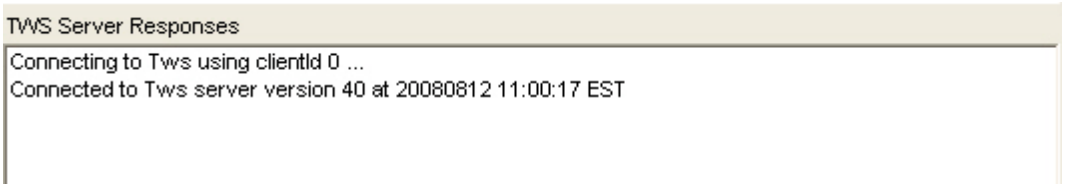
The very first thing you do with the C++ sample appliction is to connect it to a running instance of TWS. You click the Connect button to do this. This displays the Connect dialog, shown below.



- *IP Address* - This the IP address of the systemwhere Trader Workstation is installed. If TWS and the API are installed on the same computer, you can either leave this field blank, or enter 127.0.0.1 as a Trusted IP Address in TWS' API Configuration screen to indicate local host.
- *Port* - 7496 is the default port number, but you can modify this if you need to.
- *Client ID* - This is used to determine each API connection. Each connection must have a unique Client ID.

Then you can enter the *IP Address*, *Port* and *Client Id* values in the input fields of the dialog. When you click the OK button, the Connect dialog closes and a message indicating that you are connecting to TWS is displayed in the *TWS Server Responses* text panel on the sample application window. A confirmation dialog appears in TWS; click Yes to confirm that you want to connect. Note that this dialog does not appear if the IP Address of your connection is local host.

Finally, a message indicating that you have successfully connected to TWS appears in the *TWS Server Responses* text panel on the sample application window.



That's what happens on the user side of things. Now let's see what happens behind the scenes.

OnConnect()

When you click the Connect button, the OnConnect() method defined in *client2Dlg.cpp* runs. Here is what the code looks like:

```
void CClient2Dlg::OnConnect()
{
    // IMPORTANT: TWS must be running, and the "Enable Excel Integration"
    // checkbox on the "Settings" menu must be checked!

    // get connection parameters
    CDlgConnect dlg;
    if( dlg.DoModal() == IDCANCEL) {
        return;
    }

    // connect to TWS
    {
        CString displayString;
        displayString.Format( "Connecting to Tws using clientId %d ...",
            dlg.m_clientId);
        int i = m_orderStatus.AddString( displayString);
        m_orderStatus.SetTopIndex( i);
    }

    m_pClient->eConnect( dlg.m_ipAddress, dlg.m_port, dlg.m_clientId);

    if( m_pClient->serverVersion() > 0){
        CString displayString;
        displayString.Format( "Connected to Tws server version %d at %s.",
            m_pClient->serverVersion(), m_pClient->TwsConnectionTime());
        int i = m_orderStatus.AddString( displayString);
        m_orderStatus.SetTopIndex( i);
    }
}
```

OnConnect() does the following:

- Displays the Connect dialog.
- When you click the OK button in the Connect dialog, attempts to connect to TWS and displays a text message to that effect.
- Calls the C++ eConnect() method and passes the input values for the *IP Address* (dlg.m_ipAddress), *Port* (dlg.m_port) and *Client ID* (dlg.m_clientId) fields to TWS as parameters of the eCconnect() method. The important thing to remember here is that the three eConnect() parameters correspond to the three fields in the Connect dialog.

```
m_pClient->eConnect( dlg.m_ipAddress, dlg.m_port, dlg.m_clientId);
```

- If the connection to TWS is successfully established, displays the message "Connected to Tws server version at" and inserts the TWS server version and the date and time of the connection.

Disconnecting from a Running Instance of TWS

To disconnect from a running instance of TWS, click the Disconnect button in the C++ sample application. When you do this, OnDisconnect() calls the eDisconnect() C++ method, which has no parameters and disconnects the sample application from TWS. It's that simple!

Here's what OnDisconnect() looks like, with the eDisconnect() EClientSocket method highlighted in bold:

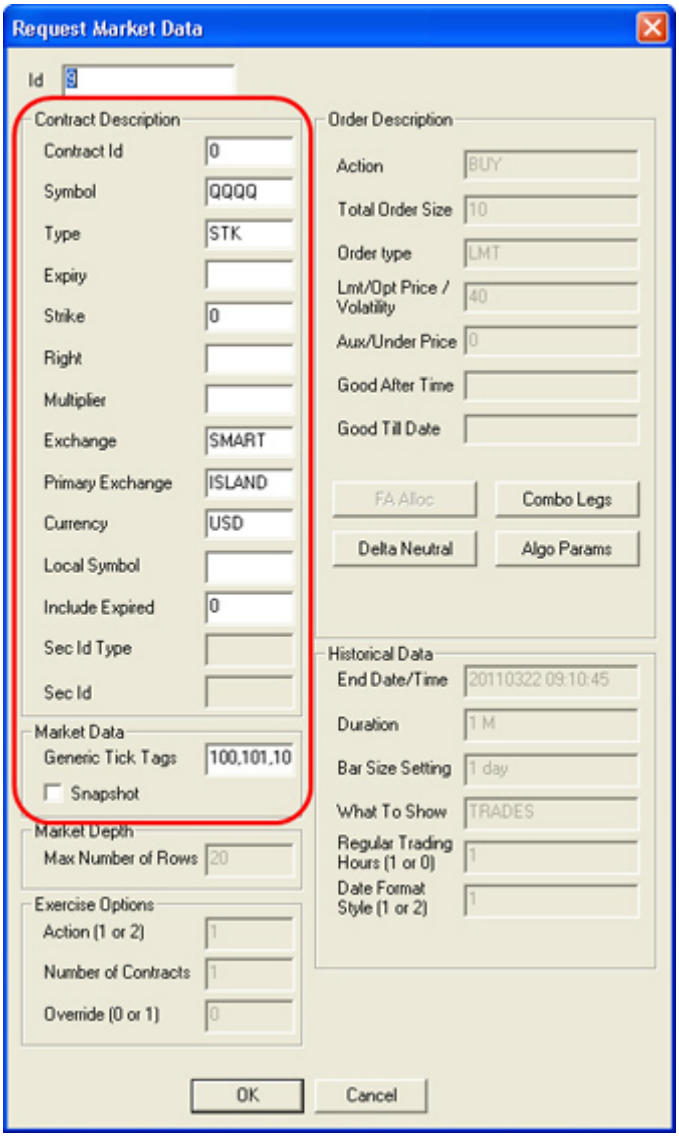
```
void CClient2Dlg::OnDisconnect()
{
    // disconnect
    m_pClient->eDisconnect();
}
```

OK, got all of that? Great! Now let's move on, and see what happens when you use the market data buttons.

Chapter 7: Requesting and Canceling Market Data

This chapter describes how the C++ sample application requests and cancels market data. You click the Req Mkt Data button to display the Request Market Data dialog, then enter information in the appropriate fields and click OK.

The following image shows the Request Market Data dialog and the fields you need to fill in to get market data.



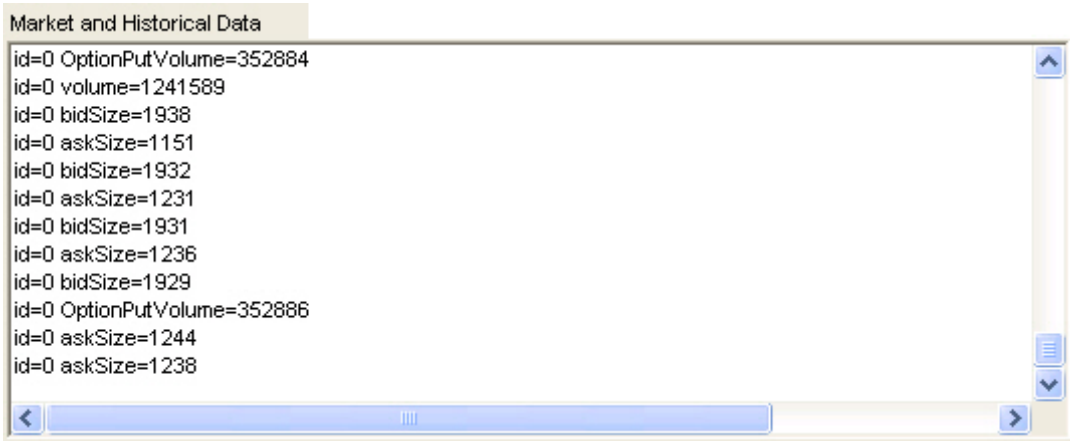
What Happens When I Click the Req Mkt Data Button?

Once you connect to TWS using the C++ sample application, you get market data by clicking the Req Mkt Data button, then entering an underlying and some other information in the Request Market Data dialog, such as symbol, type, and exchange, and clicking OK. The three sections of the dialog that you should fill in are:

- Id field - The ticker id, which must be a unique value. When the market data returns, it will be identified by this id. This is typically filled in automatically for you in our sample application, but occasionally you might get a "Duplicate Ticker ID" error message, in which case you must repeat your request for market data with an incremented ticker Id.
- Contract Description fields - These are the fields that describe the contract for which you want market data, including the symbol, type, exchange and currency. For futures and options, you fill in the other fields in the section.
- Market Data fields - There are two fields here, *Generic Tick Tags* and *Snapshot*, a check box. The snapshot option returns a single snapshot of market data if checked. Note that you cannot specify Generic Tick Tags if you select the *Snapshot* check box. You do not need to fill in the Generic Tick Tags field, bu you can if want the market data to return certain values. This field contains default values, so if you do not want to specify any generic tick tags for additional types of market data, you should delete the values from the field. Also note that if you accept the default values in the *Generic Tick Tags* field for a stock ticker, you must delete 162 (it does not apply to stocks).

For a list of all generic tick tags, see Generic Tick Types in the API Reference Guide, available on our web site.

The market data you request is displayed as scrolling lines of text in the *Market and Historical Data* text panel, as shown below.



The Symbol, Security Type, Exchange and Currency values are required for all instrument types. If your security type is STK, those four values are all you need. But if you're looking for the latest price on a Jan08 27.5 call, you need to give the method a bit more than that. The moral: be sure you include values in the appropriate fields based on what return values you want to get.

That's what happens from a user's point of view. But what's really happening?

OnReqMktData()

When you click the Req Mkt Data button, the OnReqMktData() method defined in *client2Dlg.cpp* runs. Here is what the code looks like:

```
void CClient2Dlg::OnReqMktData()
{
    // run dlg box
    m_dlgOrder->init( this, "Request Market Data", CDlgOrder::REQ_MKT_DATA,
          m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // request ticker
    m_pClient->reqMktData( m_dlgOrder->m_id, m_dlgOrder->getContract(),
       m_dlgOrder->m_genericTicks, m_dlgOrder->m_snapshotMktData);
}
```

The code samples in this book may not look exactly like the code when you view it Visual Studio. Don't worry, the code is exactly the same. We've simply formatted the code samples to fit the size of the pages in this guide.

OnReqMktData() does the following:

- Initializes the Request Market Data dialog.
- Calls the C++ method reqMktData() when the OK button is clicked.

About the Request Market Data Dialog

Before we continue with market data requests using the C++ API sample application, let's stop and take a look at the Request Market Data dialog. This is the dialog that is pictured at the beginning of this chapter and is called *DlgOrder* in the code.

As you work through the different trading tasks described in this book, you will notice that the same dialog box is used for a variety of functions. This object is used in slightly different forms to:

- request and cancel market data;
- place and cancel orders;
- request and cancel market depth;
- request contract details;

- request and cancel historical data;
- request and cancel real time bars
- exercise options.

So, for example, when you request market data, this dialog is called the Request Market Data dialog ("Request Market Data" appears in the title bar of the dialog), and only those fields required for market data requests are accessible. The fields required for the other trading tasks listed above are grayed out and unavailable.

reqMktData()

Now let's get back to requesting market data.

When you click OK in the sample application to submit a market data request, the reqMktData() method sends your market data request to TWS and, if all the entries are valid, the requested data is returned by way of the tickPrice(), tickSize(), tickGeneric(), tickOptionComputation(), tickString() and tickEFP() EWrapper functions. What's really happening though is that the reqMktData() method triggers a series of "tick" events, which return the market data from TWS. We'll look at these tick events a little later.

For now, let's find out which parameters are used for requesting market data. The reqMktData() method in our sample application code looks like this:

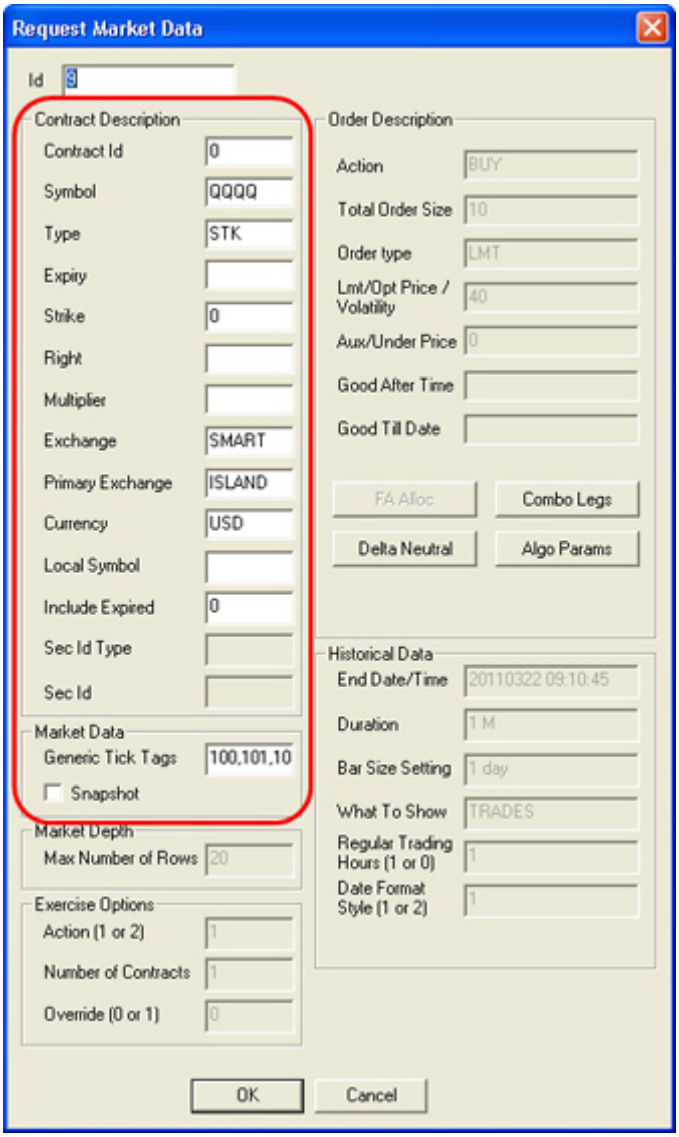
```
m_pClient->reqMktData( m_dlgOrder->m_id, m_dlgOrder->getContract(),
    m_dlgOrder->m_genericTicks, m_dlgOrder->m_snapshotMktData);
```

Parameter	Description
tickerId	The ticker id. Must be a unique value. When the market data returns, it will be identified by this tag. This is also used when canceling the market data.
contract	This object includes attributes that describe the contract.
genericTicklist	A comma delimited list of generic tick types.
snapshot	Check to return a single snapshot of market data and have the market data subscription cancel. Do not enter any genericTicklist values if you use snapshot.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

As you can see from the table above, this method has four parameters, which correspond to the fields in the Request Market Data dialog that you fill in.

Now let's take another look at the Request Market Data dialog and see how and where it relates to the reqMktData() method.

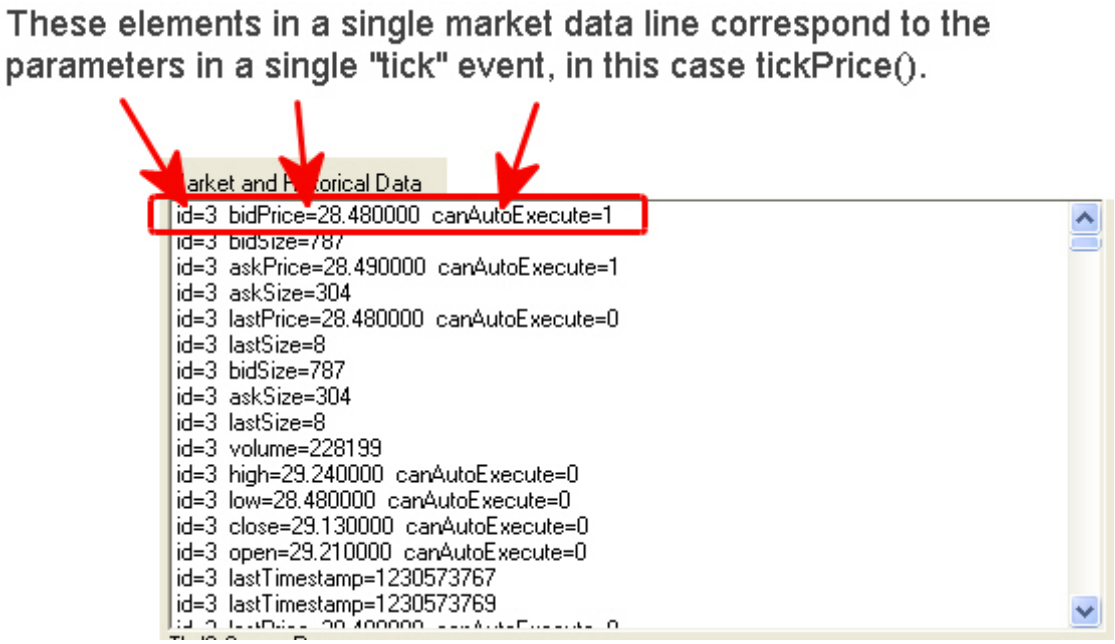


The circled sections in the picture above contain fields that correspond to the parameters in reqMktData(). This means that the values you entered in the dialog are passed to TWS by the parameters in the reqMktData() method.The *tickerId* parameter corresponds to the *Id* field in the dialog. The *snapshot* parameter is used to get a snapshot of market data; we'll describe this in more detail a little later in this chapter.

The *contract* object contains properties that correspond to the fields in the Contract Description section of the Request Market Data dialog. For a complete list of the properties in the *contract* object, see the *API Reference Guide*. You can ignore the other fields in the dialog right now because they represent parameters from different methods. Don't worry, we'll be revisiting them very soon!

C++ EWrapper Functions that Return Market Data

As we mentioned before, requested market data is returned to the sample application by way of the tickPrice(), tickSize(), tickGeneric(), tickOptionComputation(), tickString() and tickEFP() events, each of which returns a different part of the market data.



Here are the "tick" events as they appear in *client2Dlg.cpp*:

tickPrice()

```
void CClient2Dlg::tickPrice( TickerId tickerId, TickType tickType, double
price, int canAutoExecute)
{
    CString str;
    str.Format( "id=%i  %s=%f  canAutoExecute=%d",
       tickerId, (const char*)getField( tickType), price, canAutoExecute);
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

tickSize()

```
void CClient2Dlg::tickSize( TickerId tickerId, TickType tickType, int
size)
{
    CString str;
    str.Format( "id=%i  %s=%i",
       tickerId, (const char*)getField( tickType), size);
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

tickGeneric()

```
void CClient2Dlg::tickGeneric(TickerId tickerId, TickType tickType, double
value)
{
    CString str;
    str.Format( "id=%i  %s=%f",
       tickerId, (const char*)getField( tickType), value);
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

tickString()

```
void CClient2Dlg::tickString(TickerId tickerId, TickType tickType, const
CString& value)
{
    CString str;
    str.Format( "id=%i  %s=%s",
       tickerId, (const char*)getField( tickType), value);
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

tickEFP()

```
void CClient2Dlg::tickEFP(TickerId tickerId, TickType tickType, double basisPoints,
const CString& formattedBasisPoints, double totalDividends, int holdDays, const
CString& futureExpiry, double dividendImpact, double dividendsToExpiry)
{
    CString str;
    str.Format( "id=%i  %s: basisPoints=%f / %s impliedFuture=%f holdDays=%i
futureExpiry=%s dividendImpact=%f dividendsToExpiry=%f",
        tickerId, (const char*)getField( tickType), basisPoints,
formattedBasisPoints, totalDividends, holdDays, futureExpiry, dividendImpact,
dividendsToExpiry);
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

tickOptionComputation()

```
void CClient2Dlg::tickOptionComputation( TickerId tickerId, TickType
tickType, double impliedVol, double delta, double modelPrice, double
pvDividend)
{
    CString str, impliedVolStr("N/A"), deltaStr("N/A"),
modelPriceStr("N/A"), pvDividendStr("N/A");
    if (impliedVol != DBL_MAX) {
        impliedVolStr.Format("%f", impliedVol);
    }
    if (delta != DBL_MAX) {
        deltaStr.Format("%f", delta);
    }
    if (tickType != MODEL_OPTION) {
        str.Format( "id=%i %s impliedVol=%s delta=%s",
                tickerId, (const char*)getField( tickType),
impliedVolStr, deltaStr);
    } else {
        CString modelPriceStr("N/A"), pvDividendStr("N/A");
        if (modelPrice != DBL_MAX) {
            modelPriceStr.Format("%f", modelPrice);
        }
        if (pvDividend != DBL_MAX) {
            pvDividendStr.Format("%f", pvDividend);
        }
        str.Format( "id=%i  %s vol=%s delta=%s modelPrice=%s pvDividend=%s",
                tickerId, (const char*)getField( tickType),
impliedVolStr, deltaStr, modelPriceStr, pvDividendStr);
    }
    int i = m_ticks.AddString( str);

    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

The important thing to remember here is that your market data is returned from TWS by way of this series of tick events. The details of how we implemented these events in our sample application code is less important.

For more details about these events and their parameters, see the C++ Class EWrapper Functions section of the API Reference Guide.

Getting a Snapshot of Market Data

Another way to get market data from TWS to the C++ sample application is to get a snapshot of market data. A market data snapshot gives you all the market data in which you are interested for a contract for a single moment in time. What this means is that instead of watching the requested market data continuously scroll by in the *Market and Historical Data* text panel of the C++ sample application, you get a single "snapshot" of the data. This frees you from having to keep up with the scrolling data and having to cancel the market data request when you are finished.

To get snapshot market data in our C++ sample application, simply click the Req Mkt Data button, then fill in the appropriate fields in the Request Market Data dialog, and finally check the *Snapshot* check box and click OK.

You'll recall that s*napshot* is a boolean parameter of the reqMktData() method.

Canceling Market Data

When you click the Cancel Mkt Data button, the Cancel Market Data dialog appears. This is another version of the same dialog saw when we requested market data; the only difference is that when you cancel market data, all the fields in the dialog are grayed out except *Id*. Simply click OK to submit your cancellation of the market data.

So what happens in the code when you do this?

When you click the Cancel Mkt Data button, the OnCancelMktData() method in *client2Dlg.cpp* runs.

```
void CClient2Dlg::OnCancelMktData()
{
    // get ticker id
     m_dlgOrder->init( this, "Cancel Market Data", CDlgOrder::CANCEL_MKT_DATA,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // cancel market data
    m_pClient->cancelMktData( m_dlgOrder->m_id);
}
```

This method does the following:

- Displays the Cancel Market Data dialog.
- Calls the C++ method cancelMktData() when the OK button is clicked.

cancelMktData()

This method has a single parameter, *id*, which is the same ID that was specified in the reqMktData() call for market data. The cancelMktData() method is shown below as it appears in our sample application code.

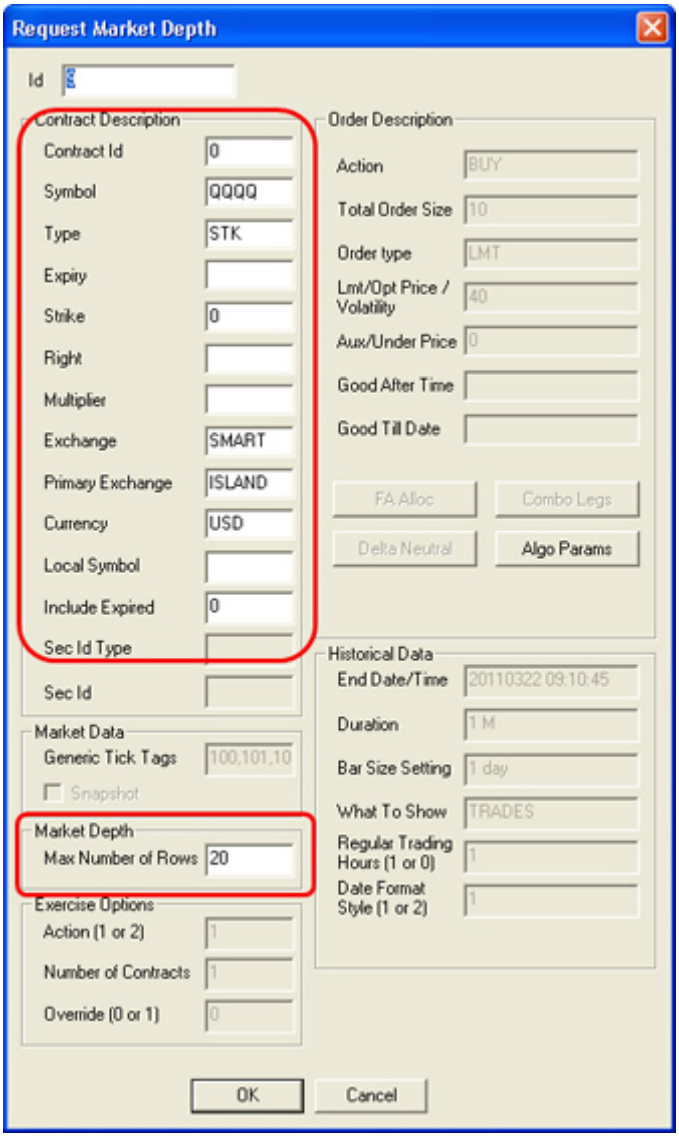
```
m_pClient->cancelMktData( m_dlgOrder->m_id);
```

Next we'll see how the C++ API handles requests for market depth.

Chapter 8 - Requesting and Canceling Market Depth

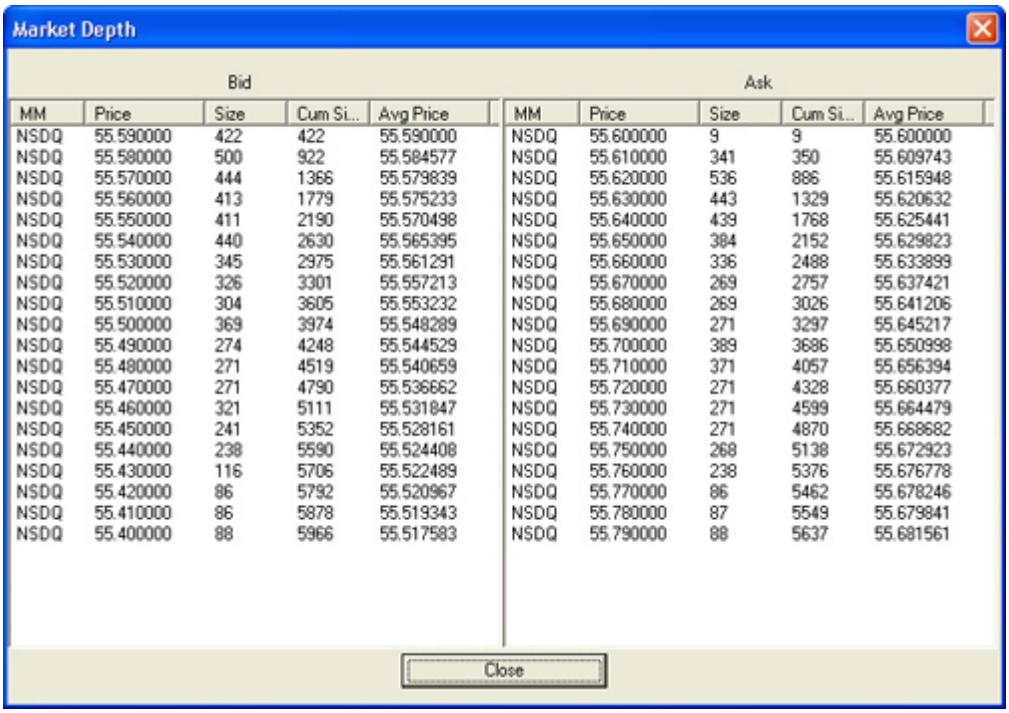
This chapter discusses the methods for requesting and canceling market depth in the C++ sample application. We'll show you the methods and parameters behind the sample application and how they call the methods in the TWS C++ API.

For requesting market depth, you need to use the highlighted fields in the Request Market Depth dialog as shown here:



What Happens When I Click the Req Mkt Depth Button?

Once you connect to TWS using the C++ sample application, you can request market depth by clicking the Req Mkt Depth button. You then enter information in the Contract Description and Market Depth fields in the Request Market Depth dialog pictured on the previous page and click OK. The market depth you request is displayed in the Market Depth dialog, an example of which is shown below.



Bid					Ask				
MM	Price	Size	Cum Si...	Avg Price	MM	Price	Size	Cum Si...	Avg Price
NSDQ	55.590000	422	422	55.590000	NSDQ	55.600000	9	9	55.600000
NSDQ	55.580000	500	922	55.584577	NSDQ	55.610000	341	350	55.609743
NSDQ	55.570000	444	1366	55.579839	NSDQ	55.620000	536	886	55.615948
NSDQ	55.560000	413	1779	55.575233	NSDQ	55.630000	443	1329	55.620632
NSDQ	55.550000	411	2190	55.570498	NSDQ	55.640000	439	1768	55.625441
NSDQ	55.540000	440	2630	55.565395	NSDQ	55.650000	384	2152	55.629823
NSDQ	55.530000	345	2975	55.561291	NSDQ	55.660000	336	2488	55.633899
NSDQ	55.520000	326	3301	55.557213	NSDQ	55.670000	269	2757	55.637421
NSDQ	55.510000	304	3605	55.553232	NSDQ	55.680000	269	3026	55.641206
NSDQ	55.500000	369	3974	55.548289	NSDQ	55.690000	271	3297	55.645217
NSDQ	55.490000	274	4248	55.544529	NSDQ	55.700000	389	3686	55.650998
NSDQ	55.480000	271	4519	55.540659	NSDQ	55.710000	371	4057	55.656394
NSDQ	55.470000	271	4790	55.536662	NSDQ	55.720000	271	4328	55.660377
NSDQ	55.460000	321	5111	55.531847	NSDQ	55.730000	271	4599	55.664479
NSDQ	55.450000	241	5352	55.528161	NSDQ	55.740000	271	4870	55.668682
NSDQ	55.440000	238	5590	55.524408	NSDQ	55.750000	268	5138	55.672923
NSDQ	55.430000	116	5706	55.522489	NSDQ	55.760000	238	5376	55.676778
NSDQ	55.420000	86	5792	55.520967	NSDQ	55.770000	86	5462	55.678246
NSDQ	55.410000	86	5878	55.519343	NSDQ	55.780000	87	5549	55.679841
NSDQ	55.400000	88	5966	55.517583	NSDQ	55.790000	88	5637	55.681561

That's what happens from a user's point of view. Let's see what's going on behind the scenes.

OnReqMktDepth()

When you click the Req Mkt Data button, the OnReqMktData() method defined in *client2Dlg.cpp* runs. Here is what the code looks like:

```
void CClient2Dlg::OnReqMktDepth()
{
    // run dlg box
     m_dlgOrder->init( this, "Request Market Depth", CDlgOrder::REQ_MKT_DEPTH,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // request ticker
    m_pClient->reqMktDepth(m_dlgOrder->m_id, m_dlgOrder->getContract(),
       m_dlgOrder->m_numRows);
    m_dlgMktDepth->DoModal();
    m_pClient->cancelMktDepth(m_dlgOrder->m_id);
}
```

OnReqMktDepth() does the following:

- Initializes the Request Market Depth dialog (which is simply another instance of *DlgOrder*).
- Calls the C++ method reqMktDepth() when the OK button is clicked.
- Initializes the Market Depth dialog, *dlgMktDepth*, which will display the market depth data returned from TWS.

The reqMktDepth() Method

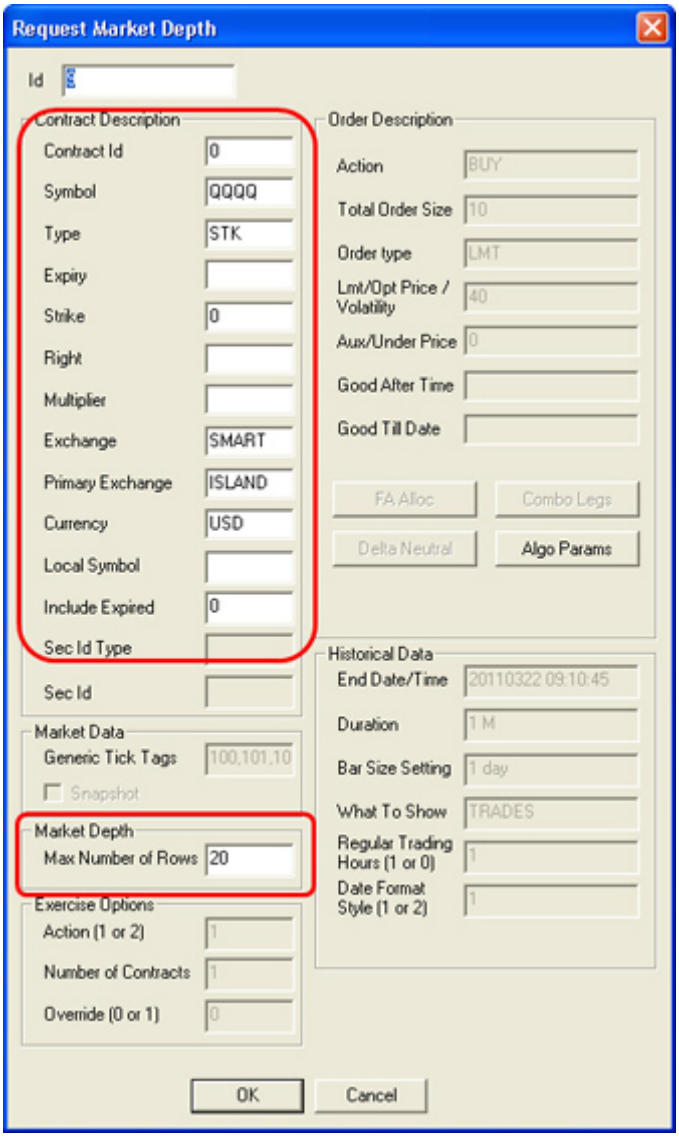
When you click OK in the sample application to submit a market depth request, the reqMktDepth() method sends your request to TWS and, if all the entries are valid, the requested data is returned by way of the updateMktDepth() and updateMktDepthL2() events. So the reqMktDepth() method triggers these two events in order to return market depth to the sample application.

Now let's see which parameters are used to request market depth. The reqMktDepth() method looks like this:

```
m_pClient->reqMktDepth(m_dlgOrder->m_id, m_dlgOrder->getContract(),
    m_dlgOrder->m_numRows);
```

Parameter	Description
tickerId	The ticker id. Must be a unique value. When the market depth data returns, it will be identified by this tag. This is also used when canceling the market depth.
contract	This object contains attributes used to describe the contract.
numRows	Specifies the number of market depth rows to return.

This table is for illustrative purposes only and is not intended to portray valid API documentation.



The circled sections in the picture above contain fields that correspond to the parameters in reqMktDepth(), so the values entered in the dialog are passed to TWS by the parameters in the reqMktDepth() method.The *tickerId* parameter corresponds to the *Id* field in the dialog, and the *numRows* parameter specifies the number of market depth rows to return to the API sample application.

The *contract* object here is the same one that is used in the reqMktData() method; it contains properties that correspond to the fields in the Contract Description section of the Request Market Depth dialog. The *contract* SocketClient property is used throughout our C++ API. For a complete list of the properties in the *contract* object, see the *API Reference Guide*.

C++ EWrapper Functions that Return Market Depth

There are two C++ EWrapper functions that return market depth: updateMktDepth() and updateMktDepthL2(). The difference between them is that updateMktDepthL2() returns LII market depth. Both functions are triggered by the reqMktDepth() method. Market depth data

is displayed in the Market Depth dialog, *DlgMktDepth*, which was initialized by the updateMktDepth() method.

The updateMktDepth() function and its parameters is shown below. You can see the data that is returned by looking at the the function's parameters in the table.

```
void CClient2Dlg::updateMktDepth(TickerId id, int position, int operation, int
side, double price, int size)
{
    m_dlgMktDepth->updateMktDepth(id, position, "", operation, side, price,
       size);
}
```

Parameter	Description
id	The ticker ID that was specified in the reqMktDepth() call.
position	Specifies the row ID of this market depth entry.
operation	Identifies how this order should be applied to the market depth. Valid values are:· • 0 = insert (insert this new order into the row identified by 'position')· • 1 = update (update the existing order in the row identified by 'position')· • 2 = delete (delete the existing order at the row identified by 'position')
side	The side of the book to which this order belongs. Valid values are: • 0 = ask • 1 = bid
price	The order price.
size	The order size.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

The updateMktDepthL2() function returns data in the same parameters as the updateMktDepth() function, and includes one additional parameter, *marketMaker*, which specifies the exchange hosting for each market depth row returned. This function only applies to customers who have subscribed to LII market data (NYSE's Open Book and NASDAQ's Total View market data subscriptions).

```
void CClient2Dlg::updateMktDepthL2(TickerId id, int position, CString marketMaker,
    int operation, int side, double price, int size)
{
    m_dlgMktDepth->updateMktDepth(id, position, marketMaker, operation, side, price,
        size);
}
```

For more details about these events and their parameters, see the C++ Class EWrapper Functions section of the API Reference Guide.

Canceling Market Depth

To cancel market depth in our C++ sample application, you first close the Market Depth dialog, then click the Cancel Mkt Depth button in the main sample application window. When you click the Cancel Mkt Depth button, the Cancel Market Depth dialog appears. Yes, this is yet another version of the same dialog we saw when we requested market data and market depth; the only difference is that when you cancel market depth, all the fields in the dialog are grayed out except *Id*. Simply click OK to submit cancel market depth.

Let's see what happens in the code when you do this.

When you click the Cancel Mkt Depth button, the OnCancelMktDepth() method in *client2Dlg.cpp* runs. The method is shown below

```
void CClient2Dlg::OnCancelMktDepth()
{
    // get ticker id
    m_dlgOrder->init( this, "Cancel Market Depth", CDlgOrder::CANCEL_MKT_DEPTH,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // cancel market data
    m_pClient->cancelMktDepth( m_dlgOrder->m_id);
}
```

OnCancelMktDepth() does the following:

- Displays the Cancels Market Depth dialog.
- Calls the C++ method cancelMktDepth() when the OK button is clicked.

The cancelMktDepth() Method

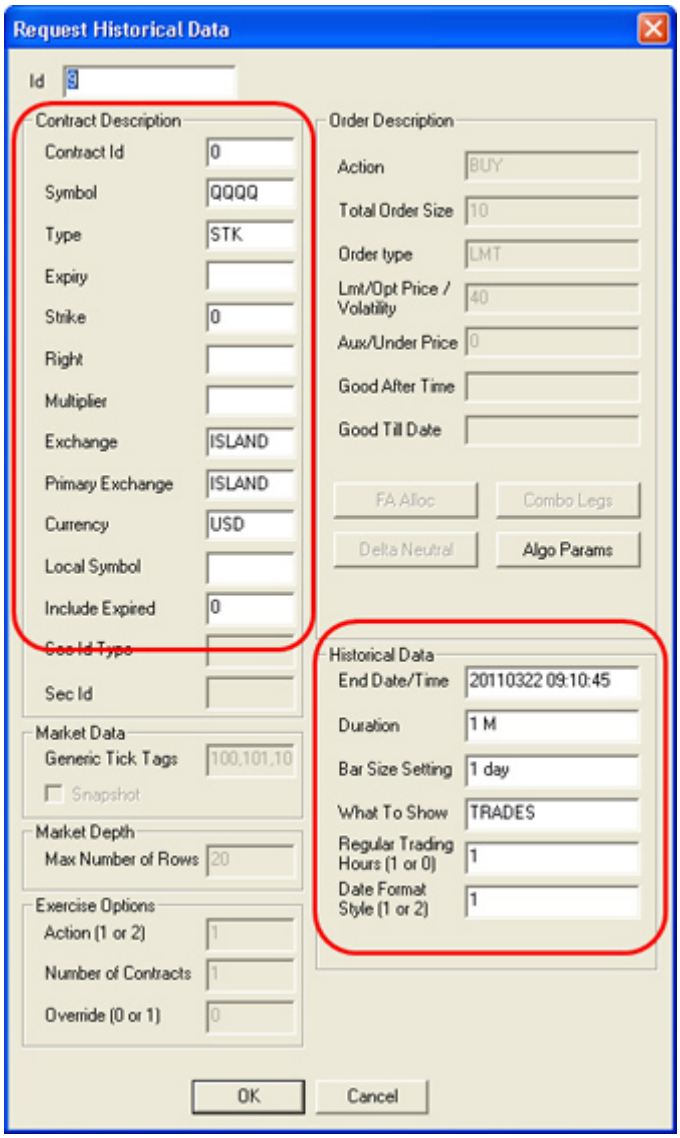
This method has a single parameter, *id*, which is the same ID that was specified in the reqMktDepth() call for market depth. The cancelMktDepth() method is shown below.

```
m_pClient->cancelMktDepth( m_dlgOrder->m_id);
```

Next we'll look at how the sample application handles another kind of data request, the historical data request.

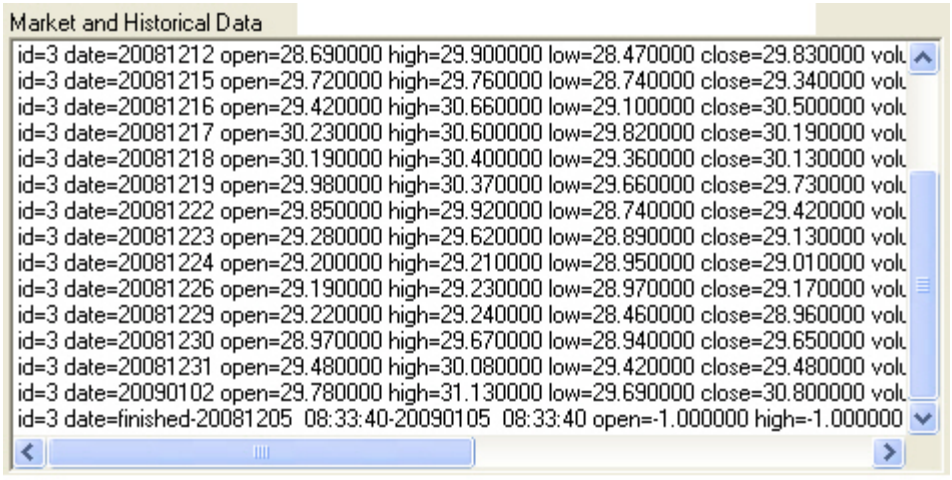
Chapter 9 - Requesting and Canceling Historical Data

This chapter describes how to request and cancel historical data in the sample application, and the API methods and parameters behind the process. For requesting historical data, you need to use the highlighted fields in the Request Historical Data dialog shown here:



What Happens When I Click the Historical Data Button?

Once you connect to TWS using the C++ sample application, you request historical data by clicking the Historical Data button, then entering contract information in the *Contract Description* section, and specifying the details of the historical data query in the *Historical Data* fields in the Request Historical Data dialog (shown on the previous page) and clicking OK. The historical data you request is displayed in the *Market and Historical Data* text panel as rows of data.



That's a simple process from a user's point of view. But what's going on behind the scenes?

OnReqHistoricalData()

Just like all the other buttons in the sample application, the Historical Data button has an event handler associated with it. When you click the Historical Data button, the OnReqHistoricalData() method defined in *client2Dlg.cpp* runs. Here is what the code looks like:

```
void CClient2Dlg::OnReqHistoricalData()
{
    m_dlgOrder->init( this, "Request Historical Data",
CDlgOrder::REQ_HISTORICAL_DATA, m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    const CString& whatToShow = m_dlgOrder->m_whatToShow;

    if ( whatToShow == "estimates" || whatToShow == "finstat" || whatToShow ==
"snapshot") {
        m_pClient->reqFundamentalData( m_dlgOrder->m_id, m_dlgOrder->getContract(),
whatToShow);
        return;
    }

    m_pClient->reqHistoricalData( m_dlgOrder->m_id, m_dlgOrder->getContract(),
        m_dlgOrder->m_backfillEndDateTime, m_dlgOrder->m_backfillDuration,
        m_dlgOrder->m_barSizeSetting, whatToShow, m_dlgOrder->m_useRTH,
        m_dlgOrder->m_formatDate);
}
```

OnReqHistoricalData() does the following:

- Initializes and displays the Request Historical Data dialog. Yes, this is another instance of the familiar *DlgOrder* object.
- If ESTIMATES, FINSTAT or SNAPSHOT are entered into the *What To Show* field in the dialog, then the event handler calls the C++ method reqFundamentalData() when the OK button is clicked.
- If anything other than ESTIMATES, FINSTAT or SNAPSHOT is entered into the *What To Show* field, the event handler calls the C++ method reqHistoricalData() when the OK button is clicked.

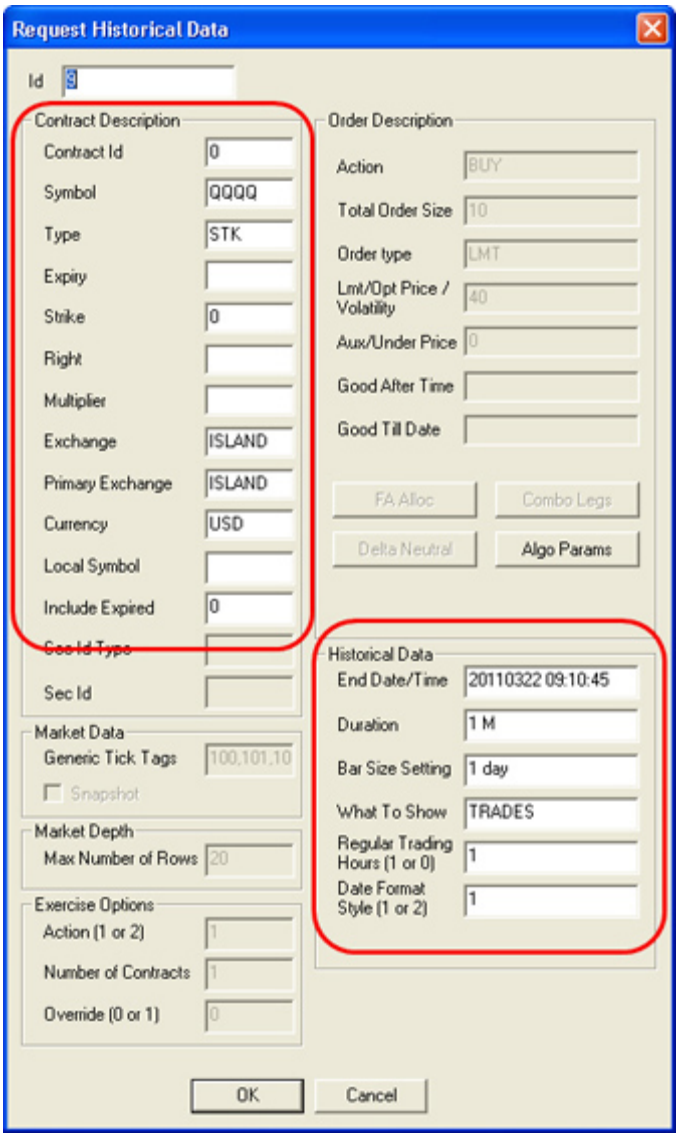
Don't worry about the reqFundamentalData() method for now. It has to do with Reuters global fundamental data, and is outside the scope of our present discussion.

The reqHistoricalData() Method

When you click OK in the sample application to request historical data, the reqHistoricalData() method sends your request to TWS and, if all the entries are valid, the requested data is returned by way of the historicalData() EWrapper function.

Now let's see which parameters are used to request historical data. The reqHistoricalData() method looks like this:

```
m_pClient->reqHistoricalData( m_dlgOrder->m_id, m_dlgOrder->getContract(),
       m_dlgOrder->m_backfillEndDateTime, m_dlgOrder->m_backfillDuration,
       m_dlgOrder->m_barSizeSetting, whatToShow, m_dlgOrder->m_useRTH,
       m_dlgOrder->m_formatDate);
```



As with the other C++ methods we've seen so far, the parameters used to request historical data correspond to the fields you complete in the Request Historical Data dialog. These fields are shown in the above screen.

The parameters are:

Parameter	Description
tickerId	The Id of the data request. Must be a unique value. When the data is received, it will be identified by this Id. This is also used when canceling the historical data request.
contract	This object includes attributes that describe the contract for which historical data is being requested.
endDateTime	This is the end data and time of the historical data request and corresponds to the field of the same name in the Historical Data section of the Request Historical Data dialog. Use the format yyyymmdd hh:mm:ss tmz, where the time zone is allowed (optionally) after a space at the end.
durationStr	This is the time span the request will cover, and is specified using the format: <integer> <unit>, i.e., 1 D, where valid units are S (seconds), D (days), W (weeks), M (months), and Y (years) . If no unit is specified, seconds are used. Also, note "years" is currently limited to one.
barSize	This parameter specifies the size of the bars that will be returned and corresponds to the field of the same name in the Request Historical Data dialog. For a complete list of the valid values for this parameter, see the reqHistoricalDataEx topic in the *API Reference Guide*.
whatToShow	This specifies the type of data to show (trades, midpoints, bids, ask, bid/ask, option implied volatility and historical volatility) and corresponds to the field of the same name in the Request Historical Data dialog.
useRTH	This parameter determines whether to return all data available during the requested time span (value = 0), or only data that falls within regular trading hours (value = 1). It corresponds to the *Regular Trading Hours* field in the dialog.
formatDate	This is the end data and time of the historical data request and corresponds to the *Date Format* field in the Request Historical Data dialog. Valid values include *1*, which returns dates applying to bars in the format: yyyymmdd{space}{space}hh:mm:dd, or *2*, which returns dates as a long integer specifying the number of seconds since 1/1/1970 GMT.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

C++ EWrapper Functions that Return Historical Data

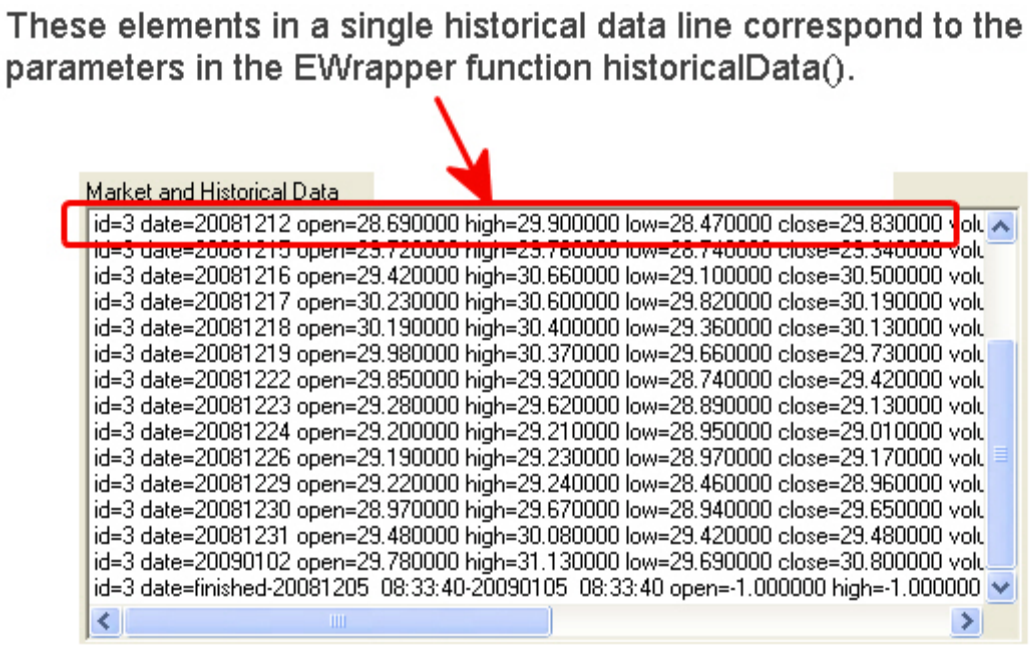
There is one C++ EWrapper function that return historical data: historicalData().This function is triggered by the reqHistoricalData() method. The historicalData() function and parameters

are shown below. You can see the data that is returned by looking at the event's parameters in the tables.

```
void CClient2Dlg::historicalData(TickerId reqId, const CString& date,
double open, double high, double low, double close, int volume, int
barCount, double WAP, int hasGaps)
{
    CString displayString;
   displayString.Format(
      "id=%d date=%s open=%f high=%f low=%f close=%f volume=%d
barCount = %d WAP=%f hasGaps=%d",
      reqId, (const char *)date, open, high, low, close, volume,
barCount, WAP, hasGaps);
    int i =  m_ticks.AddString(displayString);

   // bring into view
   int top = i - N < 0 ? 0 : i - N;
   m_ticks.SetTopIndex( top);
}
```

Parameter	Description
reqId	The ID of the request to which this bar is responding.
date	The date-time stamp of the start of the bar. The format is determined by the reqHistoricalData() formatDate parameter.
open	The bar opening price.
high	The high price during the time covered by the bar.
low	The low price during the time covered by the bar.
close	The bar closing price.
volume	The volume during the time covered by the bar.
WAP	The weighted average price during the time covered by the bar.
hasGaps	Identifies whether or not there are gaps in the data.

This table is for illustrative purposes only and is not intended to portray valid API documentation.



Historical Data Limitations

This is a good time to talk about the limitations of historical data requests. TWS API historical data requests are subject to the following limitations:

- Historical data requests can go back one full calendar year.
- Each request is restricted to duration and bar size values that return no more than 2000 bars (2000 bars per request).

All of the API technologies support historical data requests. However, requesting the same historical data in a short period of time can cause extra load on the backend and subsequently cause pacing violations. The error code and message that indicates a pacing violation is:

162 - Historical Market Data Service error message: Historical data request pacing violation

The following conditions can cause a pacing violation:

- Making identical historical data requests within 15 seconds;
- Making six or more historical data requests for the same Contract, Exchange and Tick Type within two seconds.

Also, observe the following limitation when requesting historical data:

- Do not make more than 60 historical data requests in any ten-minute period.

Canceling Historical Data

When you click the Cancel Hist. Data button, the Cancel Historical Data dialog appears. Yes, this is yet another version of the same dialog saw when we requested market data and market depth; the only difference is that when you cancel historical data, all the fields in the dialog are grayed out except *Id*. Simply click OK to cancel the historical data.

Let's see what happens in the code when you do this.

When you click the Cancel Hist. Data button, the OnCancelHistData() function defined in *client2Dlg.cpp* runs:

```
void CClient2Dlg::OnCancelHistData()
{
    m_dlgOrder->init( this, "Cancel Historical Data",
CDlgOrder::CANCEL_HISTORICAL_DATA, m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    const CString& whatToShow = m_dlgOrder->m_whatToShow;

    if( whatToShow == "estimates" || whatToShow == "finstat" || whatToShow ==
"snapshot") {
        m_pClient->cancelFundamentalData( m_dlgOrder->m_id);
        return;
    }

    m_pClient->cancelHistoricalData( m_dlgOrder->m_id);
}
```

The OnCancelHistData method does the following:

- Initializes and displays the Cancel Historical Data dialog.
- If Reuters Fundamental Data was requested, calls the cancelFundamentalData() method when the OK button is clicked.
- Calls the C++ method cancelHistoricalData() when the OK button is clicked.

The cancelHistoricalData() Method

This method has a single parameter, *tickerId*, which is the same ID that was specified in the reqHistoricalData() method that was originally called. The cancelHistoricalData() method is shown below.

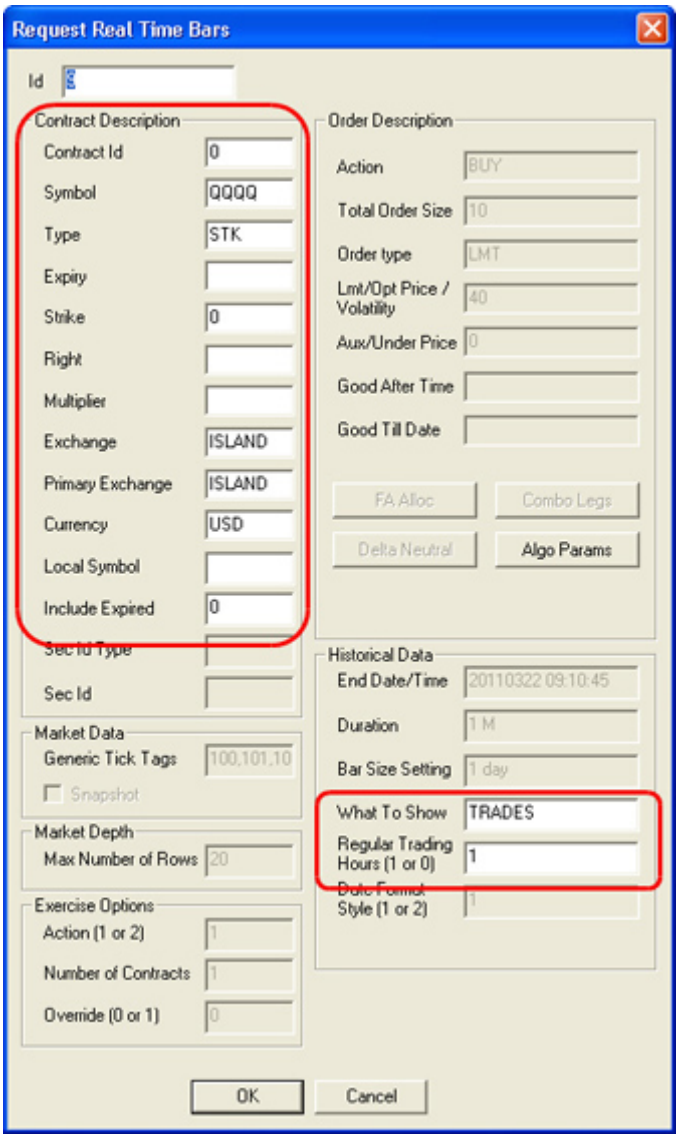
```
m_pClient->cancelHistoricalData( m_dlgOrder->m_id);
```

In the next section we'll look at how the sample application handles another kind of data request, the real-time bars request.

Chapter 10 - Requesting and Canceling Real Time Bars

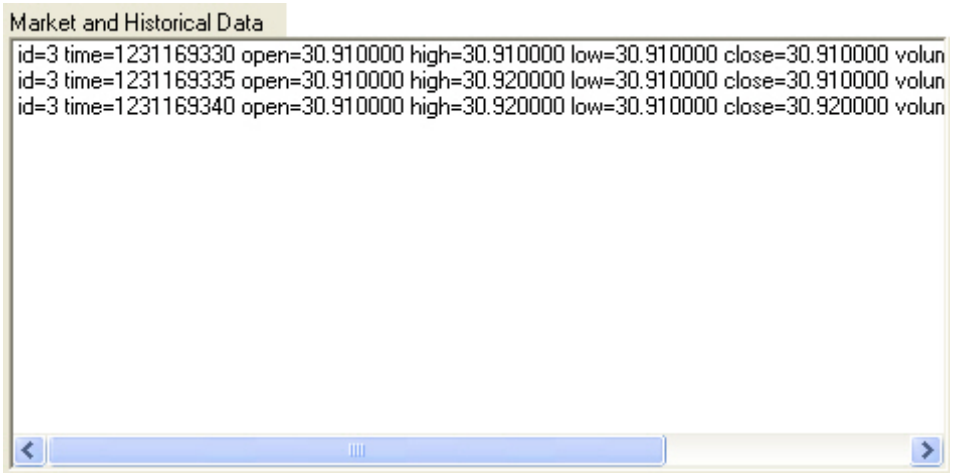
This chapter discusses the methods for requesting and canceling real time bars. Real time bars allow you to get a summary of real-time market data every five seconds, including the opening and closing price, and the high and the low within that five-second period (using TWS charting terminology, we call these five-second periods "bars"). You can also get data showing trades, midpoints, bids or asks. In this chapter, you'll learn about the methods and parameters behind the process of requesting real-time bars.

For requesting real time bars, you need to use the fields circled in the Request Real Time Bars dialog shown below.



What Happens When I Click the Real Time Bars Button?

You request real time bars by, well, clicking the Real Time Bars button, then entering contract information in the *Contract Description* section, and specifying the details of the real-time bar data query in two fields in the *Historical Data* section in the Request Real Time Bars dialog (shown on the previous page) and clicking OK. The data you request is displayed in the *Market and Historical Data* text panel as rows of data.



Once again, this is a simple process from a user's point of view. Let's take a look at what's happening in the code when you request real time bars.

OnReqRealTimeBars()

Just like all the other buttons in the sample application, the Real Time Bars button has an "On" method associated with it. When you click the button, OnReqRealTimeBars(), defined in *client2Dlg.cpp*, runs. Here is what the code looks like:

```
void CClient2Dlg::OnReqRealTimeBars()
{
    m_dlgOrder->init( this, "Request Real Time Bars", CDlgOrder::REQ_REAL_TIME_BARS,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    m_pClient->reqRealTimeBars( m_dlgOrder->m_id, m_dlgOrder->getContract(),
       5 /* TODO: parse and use m_dlgOrder->m_barSizeSetting) */,
        m_dlgOrder->m_whatToShow, m_dlgOrder->m_useRTH > 0);
}
```

OnReqRealTimeBars() does the following:

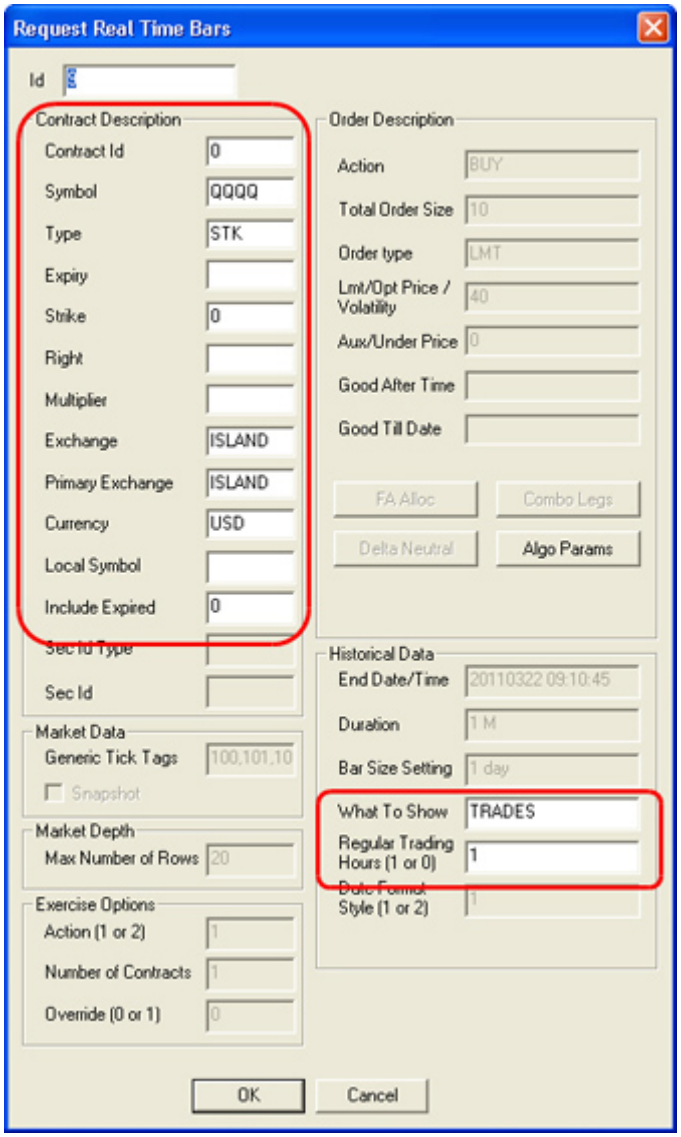
- Initializes and displays the Request Real Time Bars dialog, which is yet another instance of the *DlgOrder* object.
- Calls the C++ method reqRealTimeBars() when the OK button is clicked.

The reqRealTimeBars() Method

When you click OK in the sample application to request real time bars, the C++ API EClientSocket method reqRealTimeBars() sends your request to TWS and, if all the entries are valid, the requested data is returned by way of the realTimeBar() EWrapper function.

Now let's see which parameters are used to request real time bars. The reqRealTimeBars() method looks like this:

```
m_pClient->reqRealTimeBars( m_dlgOrder->m_id, m_dlgOrder->getContract(),
5 /* TODO: parse and use m_dlgOrder->m_barSizeSetting) */,
m_dlgOrder->m_whatToShow, m_dlgOrder->m_useRTH > 0);
```



As with the other C++ methods we've seen so far, the parameters used to request real time bars correspond to the fields you complete in the Request Real Time Bars dialog. The fields used to request real time bars are some of the same fields you used to request historical data.

Parameter	Description
tickerId	The Id for the request. Must be a unique value. When the data is received, it will be identified by this Id. This is also used when canceling the real-time bars data request.
contract	This object contains a description of the contract for which real time bars are being requested.
barSize	This parameter specifies the size of the bars that will be returned. Currently only 5 second bars are supported, if any other value is used, an exception will be thrown.
whatToShow	This specifies the type of data to show (trades, bid, ask, or midpoints), and corresponds to the field of the same name in the Historical Data section of the Request Real Time Bars dialog.
useRTH	This parameter determines whether to return all data available during the requested time span (value = 0), or only data that falls within regular trading hours (the value = 1). It corresponds to the Regular Trading Hours field in the Historical Data section of the Request Real Time Bars dialog.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

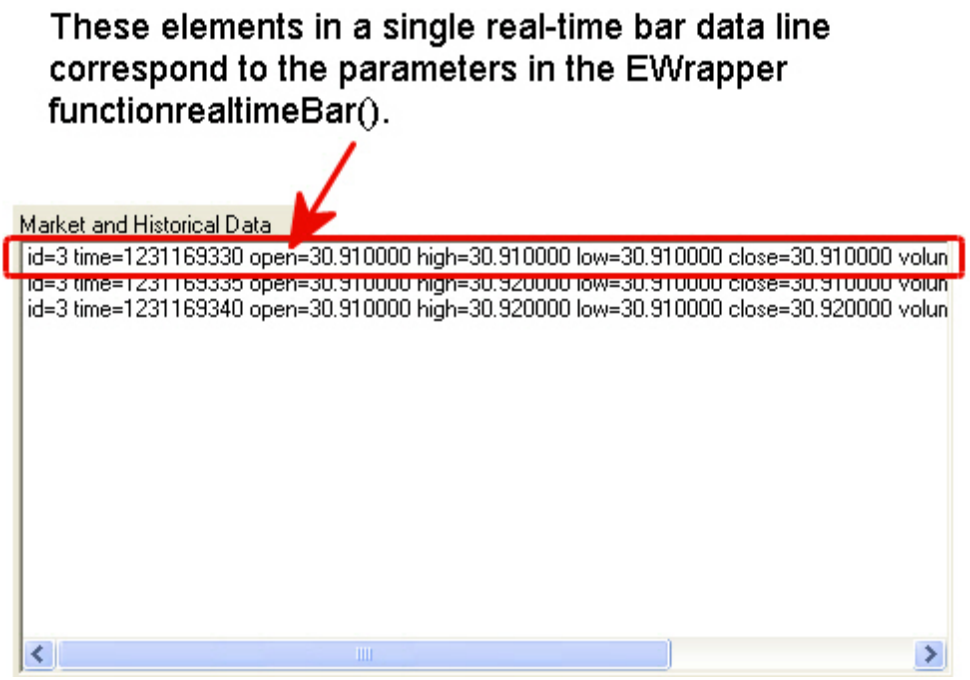
C++ EWrapper Functions that Return Real Time Bars

There is one function that return historical data: realtimeBar().This event is triggered by the reqRealTimeBars() method. The realtimeBar() event and its parameters are shown below. You can see the data that is returned by looking at the the event's parameters in the tables.

```
Sub realtimeBar(ByVal tickerId As Integer, ByVal time As Integer,
ByVal open As Double, ByVal high As Double, ByVal low As Double, ByVal
close As Double, ByVal volume As Integer, ByVal WAP As Double, ByVal
Count As Integer)
```

Parameter	Description
reqId	The ticker Id of the request to which this bar is responding.
time	The date-time stamp of the start of the bar. The format is determined by the reqHistoricalData() formatDate parameter.
open	The bar opening price.
high	The high price during the time covered by the bar.
low	The low price during the time covered by the bar.
close	The bar closing price.
volume	The volume during the time covered by the bar.
wap	The weighted average price during the time covered by the bar.
count	When TRADES historical data is returned, represents the number of trades that occurred during the time period the bar covers.

This table is for illustrative purposes only and is not intended to portray valid API documentation.



Canceling Real Time Bars

When you click the Can Real Time Bars button, the OnCancelRealTimeBars() method defined in *client2Dlg.cpp* runs:

```
void CClient2Dlg::OnCancelRealTimeBars()
{
    m_dlgOrder->init( this, "Cancel Real Time Bars",
CDlgOrder::CANCEL_REAL_TIME_BARS, m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    m_pClient->cancelRealTimeBars( m_dlgOrder->m_id);
}
```

The cmdCancelRealTimeBars_Click button event handler does the following:

- Initializes and displays the Cancel Real Time Bars dialog. Yes, this is another instance of the same *DlgOrder* object we've seen before.
- Calls the C++ method cancelRealTimeBars() when the OK button is clicked.

The cancelRealTimeBars() Method

This method has a single parameter, *tickerId*, which is the same ID that was specified in the reqRealTimeBars() call for real time bars. The cancelRealTimeBars() method is shown below.

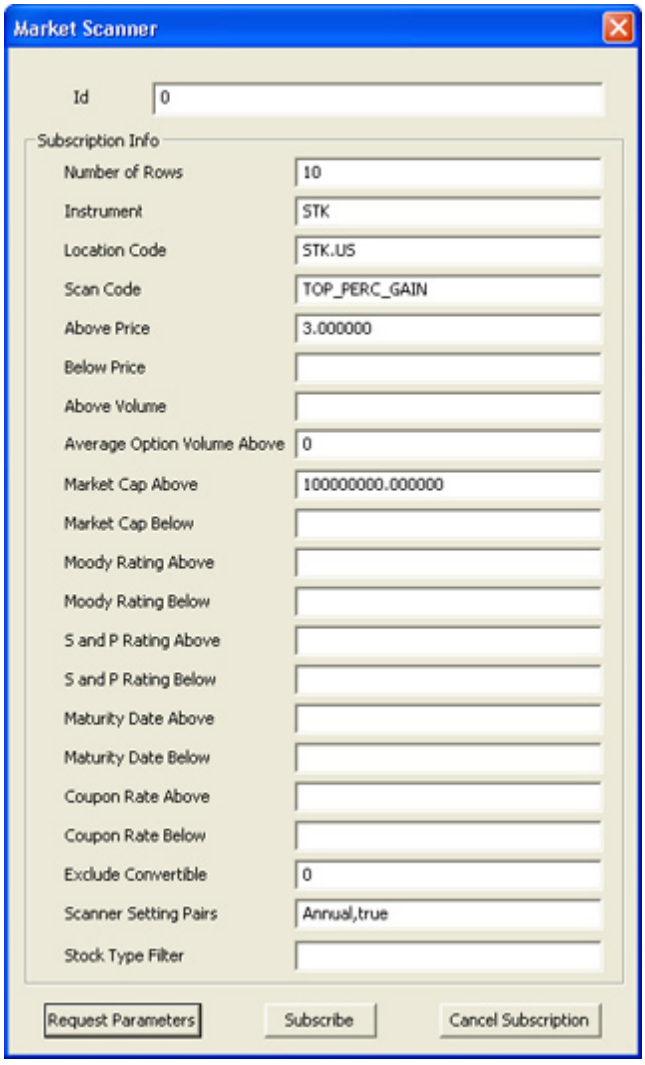
```
m_pClient->cancelRealTimeBars( m_dlgOrder->m_id);
```

As you can see from the similarity in the code for the trading tasks you've looked at so far, we've tried to make the C++ API methods and events as constent and easy to understand as possible.

The next chapter takes a looks at how to run a market scanner using the C++ API sample application.

Chapter 11 - Subscribing to and Canceling Market Scanner Subscriptions

This chapter describes the methods and events used for requesting market scanner parameters, subscribing to a market scanner, and canceling a subscription to a market scanner in the C++ sample application. When you click the Market Scanner button in the sample application, the Market Scanner dialog opens, instead of the standard Request/Cancel dialog we've seen for the other buttons on the sample application.



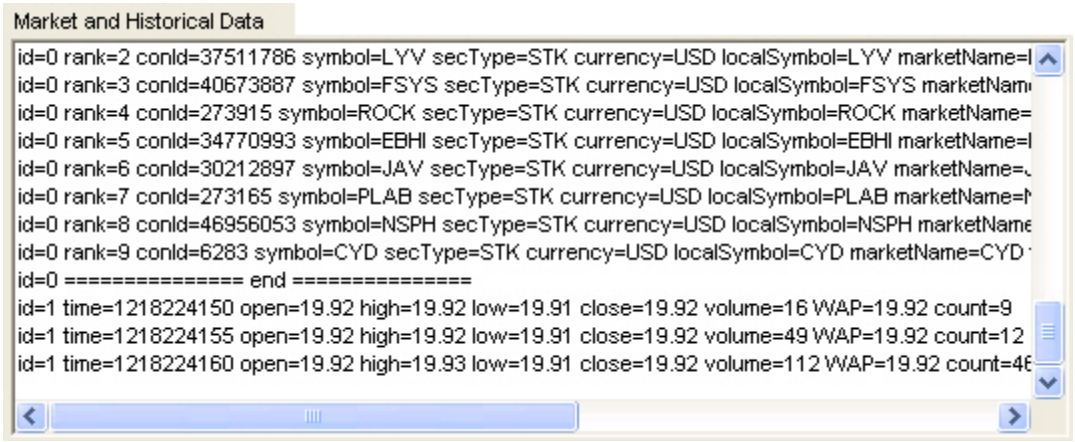
What Happens When I Click the Market Scanner Button?

You can do two things related to market scanners in the C++ sample application:

- You can subscribe to a market scanner.
- You can request scanner parameters via an XML document, which is displayed in the sample application. This XML document describes the valid parameters that a scanner subscription can have.

To perform either of these tasks, you click the Market Scanner button, then enter information in the Market Scanner dialog and click the appropriate button. For a market scan, fill in as many of the fields in the dialog as you need (especially the *Scan Code*!), then click the Subscribe button. To request available scan parameters, click the Request Parameters button.

Market scan results are displayed in the *Market and Historical Data* text panel as shown below. Market scanner parameters (the XML file) are displayed in the *TWS Server Responses* text panel.



This is a fairly simple process from a user's point of view. There's more going on behind the scenes, however, so let's take a look.

OnMarketScanner()

Just like all the other buttons in the sample application, the Market Scanner button has an "On" method associated with it. When you click the button, OnMarketScanner(), defined in *client2Dlg.cpp*, runs. Here is what the code looks like:

```
void CClient2Dlg::OnMarketScanner()
{
    CDlgScanner dlgScanner(m_scannerSubscr.get(), m_pClient);
    dlgScanner.DoModal();
}
```

OnMarketScanner() really does only one thing: it displays the Market Scanner dialog, or as its known in the code, DlgScanner. To find out how the sample application actually gets the scanner parameters and subscribes to a market scanner, we have to look at the Market Scanner dialog code in *DlgScanner.cpp*.

Market Scanner Dialog

The process of filling in the fields of the Market Scanner dialog is pretty straightforward. However, the code for the Market Scanner dialog does a few things in which we're interested:

- Just as we saw in the the *client2Dlg.cpp* source file, *DlgScanner.cpp* includes a MESSAGE MAP section that maps the buttons in the Market Scanner dialog to related "On" methods. These "On" methods are not part of the C++ API, but are part of the code required to run the C++ API sample application.
- Runs the OnRequestParameters() method when you click the Request Parameters button in the Market Scanner dialog.
- Runs the OnSubscribe() method when you click the Subscribe button.
- Runs the OnCancelSubscription() method when you click the Cancel Subscription button. (We'll learn more about how to cancel a market scanner subscription later in this chapter.)

OnRequestParameters() calls the reqScannerParameters() EClientSocket method when you click the Request Parameters button.

OnSubscribe() calls the reqScannerSubscription() EClientSocket method when you click the Subscribe button.

Turn the page to learn about these two methods!

Requesting Scanner Parameters

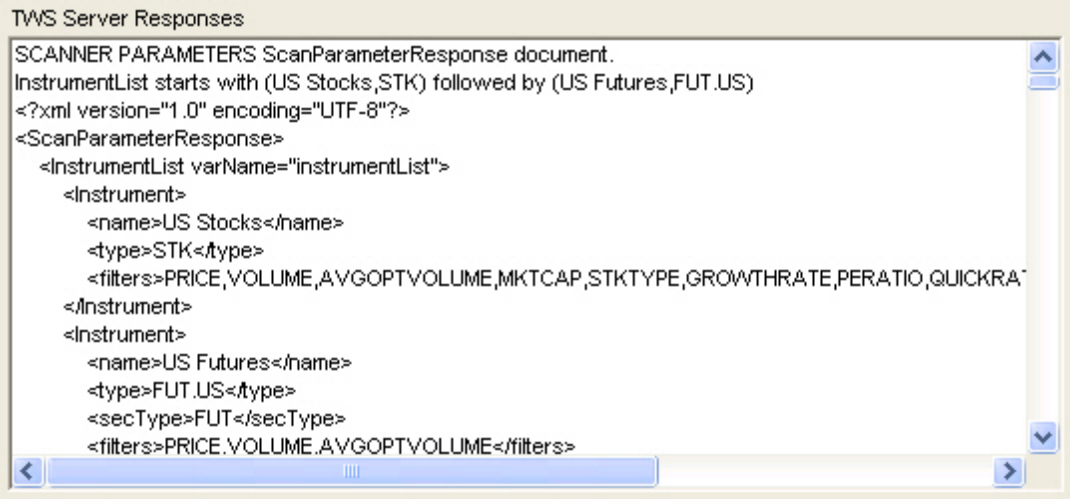
As we noted earlier, when you click the Request Parameters button in the Market Scanner dialog, the OnRequestParameters() method calls the reqScannerParameters() method, which requests an XML string that describes all possible market scans and their available parameters.

```
void CDlgScanner::OnRequestParameters() {
    OnOK();
    m_client->reqScannerParameters();
}
```

As you can see, this method has no parameters. The XML document containing the scanner parameters is returned from TWS by the scannerParameters() EWrapper function, which is defined in *client2Dlg.cpp* and is shown below.

```
void CClient2Dlg::scannerParameters(const CString &xml)
{
    static CString myTitle("SCANNER PARAMETERS: ") ;
     DisplayMultiline(m_orderStatus, myTitle, xml);
}
```

The scannerParameters() function has a single parameter, *xml*, which you have probably figured out is the XML document that contains the scanner parameters, including all of the current scan codes that are required to subscribe to the market scanners. This is the XML document that is displayed in the *TWS Server Responses* text panel of the sample application and a portion of which is shown below.



Subscribing to a Market Scanner

When you click the Subscribe button in the Market Scanner dialog, the OnSubscribe() method calls the reqScannerSubscription() method, which is shown below along with its parameters,

```
void CDlgScanner::OnSubscribe() {
    OnOK();
    m_client->reqScannerSubscription(m_id, *m_subscription);
}
```

Parameter	Description
tickerId	The ticker id. Must be a unique value. When the market scanner results are returned, they will be identified by this tag. This is also used when unsubscribing from the scanner.
subscription	This object contains the scanner subscription parameters.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

As you can see from the table above, this method has two parameters. The *tickerId* parameter corresponds to the *ID* field in the Market Scanner dialog. The *subscription* parameter is actually another EClientSocket property called ScannerSubscription, and it contains the properties that correspond to the fields in the Market Scanner dialog, such as *Instrument* and *Scan Code*.

For a complete list of the properties in the *subscription* structure, see the *API Reference Guide*.

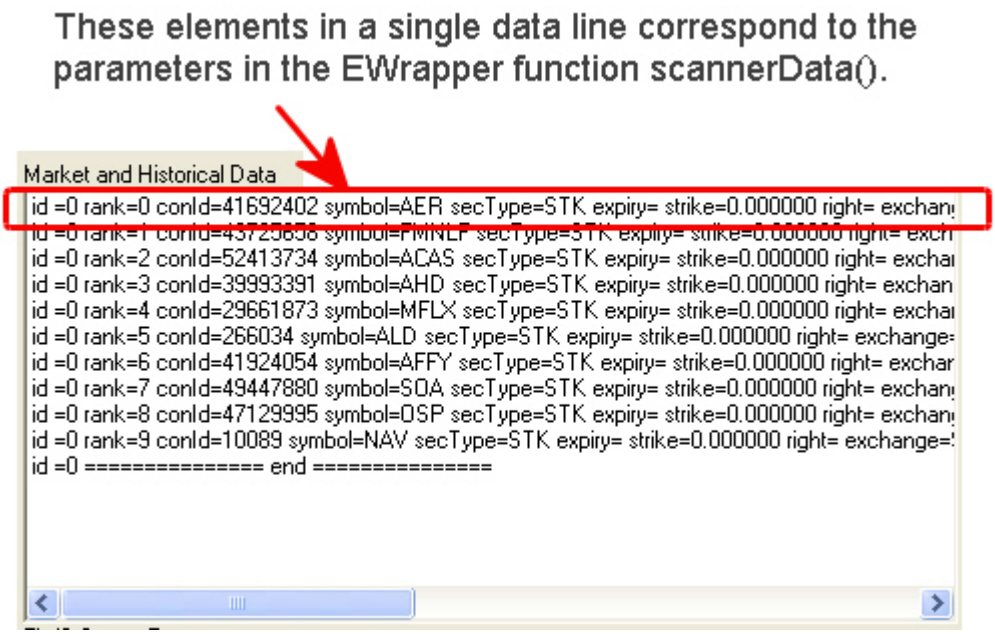
C++ EWrapper Functions that Return Market Scanner Results

The market scan results are returned by the scannerData() EWrapper function, which is shown below along with its parameters:.

```
void CClient2Dlg::scannerData(int reqId, int rank, const ContractDetails
&contractDetails, const CString &distance, const CString &benchmark, const
CString &projection, const CString &legsStr) {
    Contract contract = contractDetails.summary;
    // create string
    CString str;
    str.Format("id =%i rank=%i conId=%i symbol=%s secType=%s expiry=%s
strike=%f right=%s exchange=%s currency=%s localSymbol=%s marketName=%s
tradingClass=%s distance=%s benchmark=%s projection=%s legsStr=%s",
        reqId,
        rank,
       contract.conId,
        contract.symbol,
        contract.secType,
        contract.expiry,
        contract.strike,
        contract.right,
        contract.exchange,
        contract.currency,
        contract.localSymbol,
        contractDetails.marketName,
        contractDetails.tradingClass,
        distance,
        benchmark,
        projection,
       legsStr);
    int i =  m_ticks.AddString(str);

    // bring into view
    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

Parameter	Description
reqId	The ticker ID of the request to which this row is responding.
rank	The ranking within the response of this bar.
contractDetails	This object contains a full description of the contract.
distance	Varies based on query.
benchmark	Varies based on query.
projection	Varies based on query.
legsStr	Describes combo legs when scan is returning EFP

The *contractDetails* parameter is another EClientSocket property and it contains information about the contract whose data is included in the market scanner results. For more information about this and other EClientSocket properties, see the *API Reference Guide*.



The scannerDataEnd() Function

There is one additional event used in conjunction with scanner subscriptions: scannerDataEnd(). This function is called after a full snapshot of a scanner window has been received and functions as a sort of end tag. It helps define the end of one scanner snapshot and the beginning of the next. scannerDataEnd() has one parameters: *reqId*, which is the ID of the market scanner request being closed by this parameter.

The scannerDataEnd() function is shown below.

```
void CClient2Dlg::scannerDataEnd(int reqId)
{
    // create string
    CString str;
    str.Format("id =%i ========== end ==========", reqId);
     int i =  m_ticks.AddString(str);

    // bring into view
    int top = i - N < 0 ? 0 : i - N;
    m_ticks.SetTopIndex( top);
}
```

Cancel a Market Scanner Subscription

To cancel a market scanner subscription in the C++ sample application, first click Market Scanner button, then click the Cancel Subscription button in the Market Scanner dialog. When you click this button, the OnCancelSubscription() method defined in *DlgScanner.cpp* runs.

```
void CDlgScanner::OnCancelSubscription() {
    OnOK();
    m_client->cancelScannerSubscription(m_id);
}
```

The OnCancelSubscription() method calls the cancelScannerSubscription() EWrapper function, shown below, which cancels the market scanner subscription.

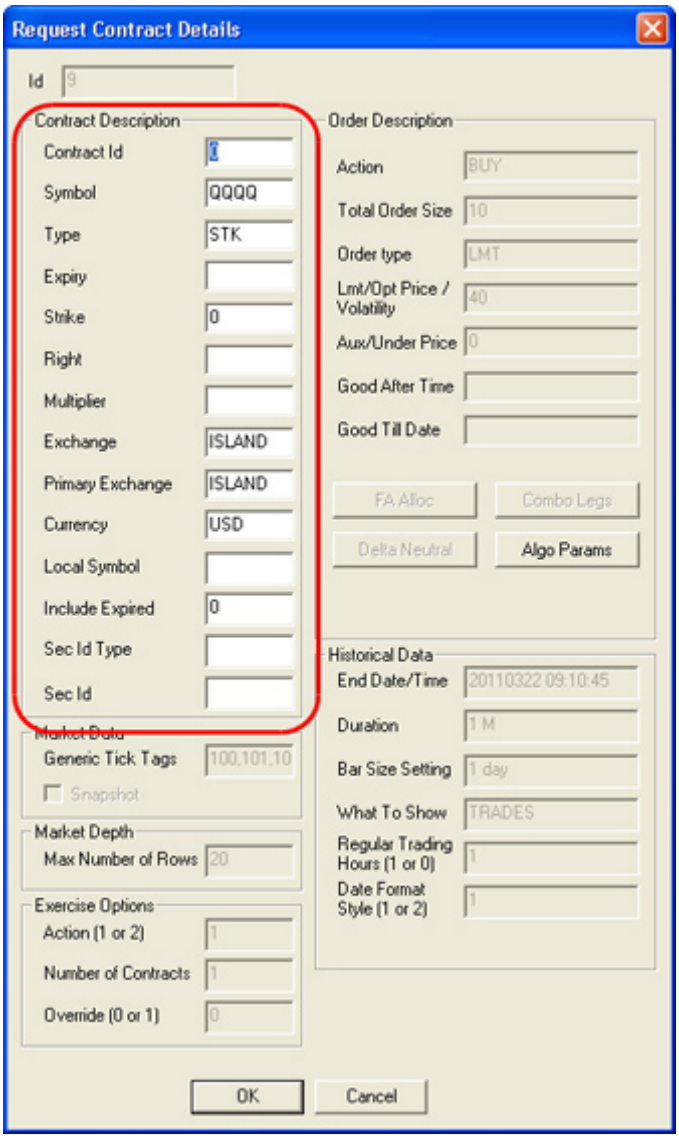
```
m_client->cancelScannerSubscription(m_id);
```

The next chapter takes a look at one additional type of data that you can get from TWS via our C++ API: contract data.

Chapter 12: Requesting Contract Data

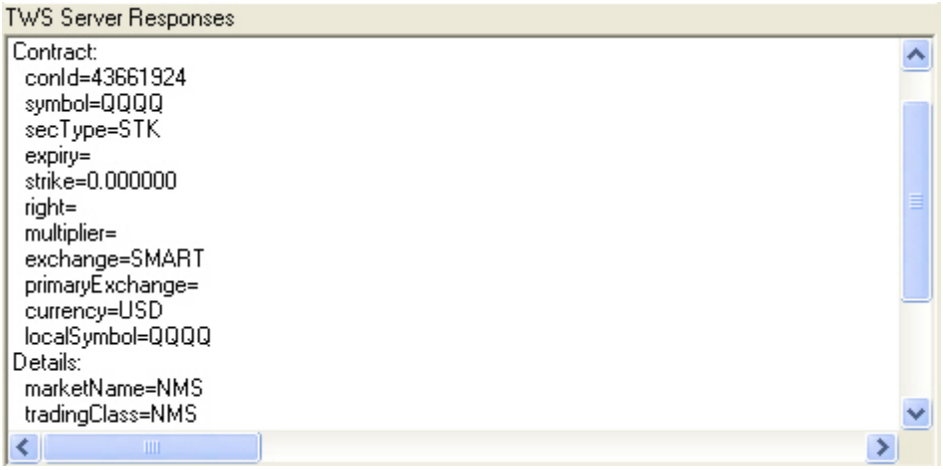
This chapter shows you how to request contract data, including details such as the local symbol, conid, trading class, valid order types, and exchanges. We'll walk you through everything that happens from the time you click the Req Contract Data button in the sample application, to the moment you're taking in the fascinating details of your desired contract. It all happens fast, so pay attention!

To request contract data using the C++ sample application, you'll need to enter data in the fields circled in the Request Contract Details dialog pictured below. The Request Contract Details dialog appears when you click the Req Contract Data button.



What Happens When I Click the Req Contract Data Button?

In the C++ sample application, you request contract details by clicking the Req Contract Data button, then entering an underlying and other information in the *Contract Description* fields of the Request Contract Details dialog, shown on the previous page. When you click OK, the contract data you requested are displayed in the *TWS Server Responses* text panel, as shown here:



Let's take a look at the code behind this simple process.

OnReqContractDetails()

When you click the Req Contract Data button, the OnReqContractDetails() method defined in *client2Dlg.cpp* runs. Here is what that method looks like:

```
void CClient2Dlg::OnReqContractDetails()
{
    // run dlg box
     m_dlgOrder->init( this, "Request Contract Details",
CDlgOrder::REQ_CONTRACT_DETAILS, m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // request contract details
    m_pClient->reqContractDetails( m_dlgOrder->m_id,
       m_dlgOrder->getContract());

}
```

OnReqContractDetails() does the following:

- Initializes and displays the Request Contract Details dialog, which is another instance of our old friend *DlgOrder*.
- Calls the C++ method reqContractDetails() when the OK button is clicked.

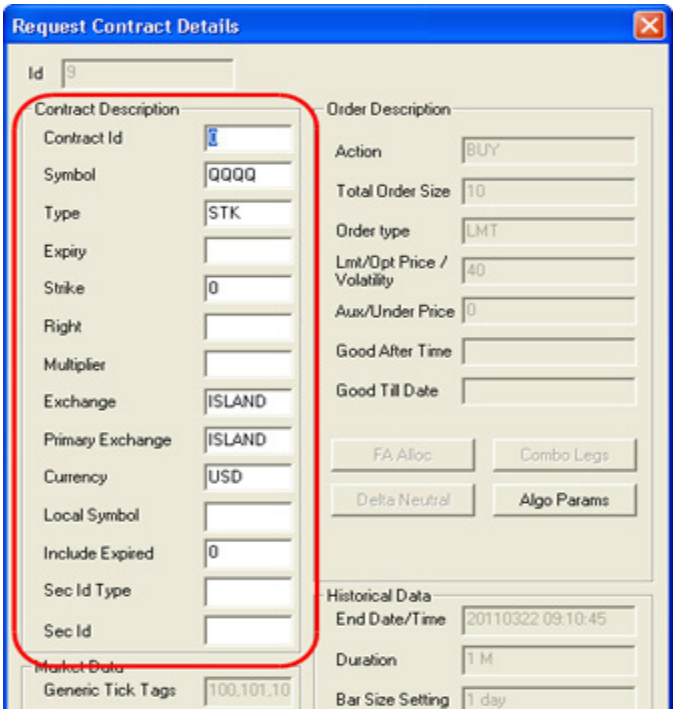
The reqContractDetails() Method

Your contract data request is passed to TWS via the reqContractDetails() method.

```
m_pClient->reqContractDetails( m_dlgOrder->m_id,
m_dlgOrder->getContract());
```

Parameter	Description
reqId	The ID of the data request. Ensures that responses are matched to requests if several requests are in process.
contract	This object includes attributes that describe the contract.

This method contains two parameters, *reqId* and *Contract*. *reqId* passes the ID of the data request to TWS and ensures that responses are matched to requests if several requests are in process. If you recall from earlier chapters, the *Contract* parameter is an EClientSocket property that contains all the attributes used to describe the requested contract, which in this case means all the values you entered in the *Contract Description* section of the Request Contract Details dialog.

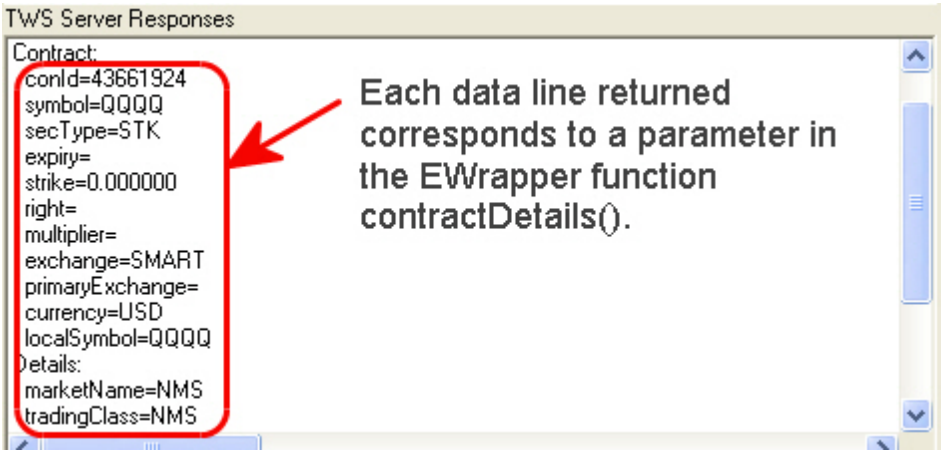


C++ EWrapper Functions that Return Contract Details

The actual contract data is returned from TWS via the contractDetails() EWrapper function.

```
void CClient2Dlg::contractDetails( int reqId, const ContractDetails
&contractDetails)
```

This function contains two parameters, *reqId* and *contractDetails*. Just as in the method that requested the contract data, the *reqID* parameter here contains the ID of the data request to ensure matching with the correct request. The *contractDetails* parameter is another EClientSocket property (like *Contract or ScannerSubscription*) that contains all the attributes used to describe the requested contract, including the valid order types and exchanges on which the requested contract can be traded.



For more details about the *contractDetails* structure, see the *API Reference Guide*.

The contractDetailsEnd() Function

There is one additional contract data EWrapper function used in the C++ API, the contractDetailsEnd() event. This event has one parameter, *reqId*, and is called once all contract details for a given request are received. This defines the end of an option chain.

```
void CClient2Dlg::contractDetailsEnd( int reqId)
{
    CString str;
    str.Format("id =%i =============== end ===============", reqId);

    int i = m_orderStatus.AddString(str);

    // bring into view
    int top = i - N < 0 ? 0 : i - N;
    m_orderStatus.SetTopIndex(top);
}
```

contractDetailsEnd() displays the end marker for the contract information displayed in the *TWS Server Responses* text panel of the sample application main window.

This concludes our discussion of market and contract data-related trading tasks. The next section describes how to place orders and exercise options using the sample application and C++ API.

5

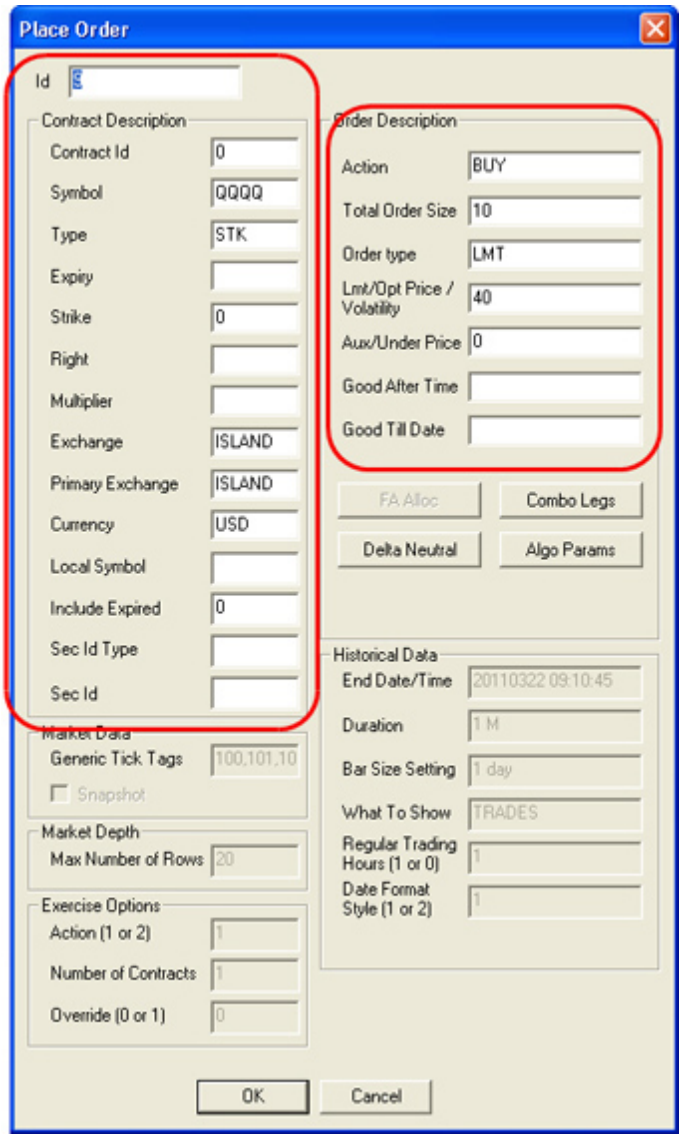
Orders and Executions

This section describes how the C++ API sample application handles orders. We'll show you the methods, events and parameters behind such trading tasks as placing and canceling orders, exercising opions and viewing open orders and executions.

Here's what you'll find in this section:

Chapter 13: Placing and Canceling an Order

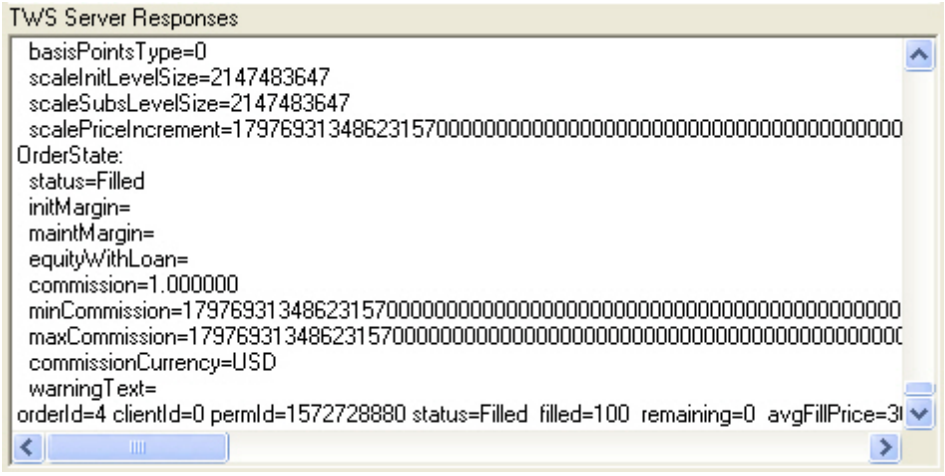
In this chapter, we describe what happens when you place and cancel an order. When you click the Place Order button, another version of the standard Request/Cancel dialog (the dlgOrder object) opens. To place an order, you fill in the fields circled in the dialog as shown below.



What Happens When I Place an Order?

Let's take a look at what happens when you place an order. First we'll look at how a typical user would place an order using the C++ sample application, then we'll look at the code behind that process.

When you click the Place Order button, the Place Order dialog appears. As we mentioned earlier, this is yet another version of the *DlgOrder* object. You enter the contract information in the *Contract Description* fields, enter the order information in the *Order Description* fields, then click the OK button to place the order. Your order information is displayed in the *TWS Server Responses* text panel on the sample application window, along with the execution details and the order status.



There are additional order options available in the C++ sample application that are supported by the API. You can enter combo orders or Algo parameters for IBAlgo orders. These options are described later in this chapter.

That's pretty straightforward. Now let's see how the code works during this process.

OnPlaceOrder()

When you click the Place Order button, the OnReqContractDetails() method defined in *client2Dlg.cpp* runs. Here is what that method looks like:

```
void CClient2Dlg::OnPlaceOrder()
{
    placeOrder(/* whatIf */ false);
}
```

The placeOrder() Method

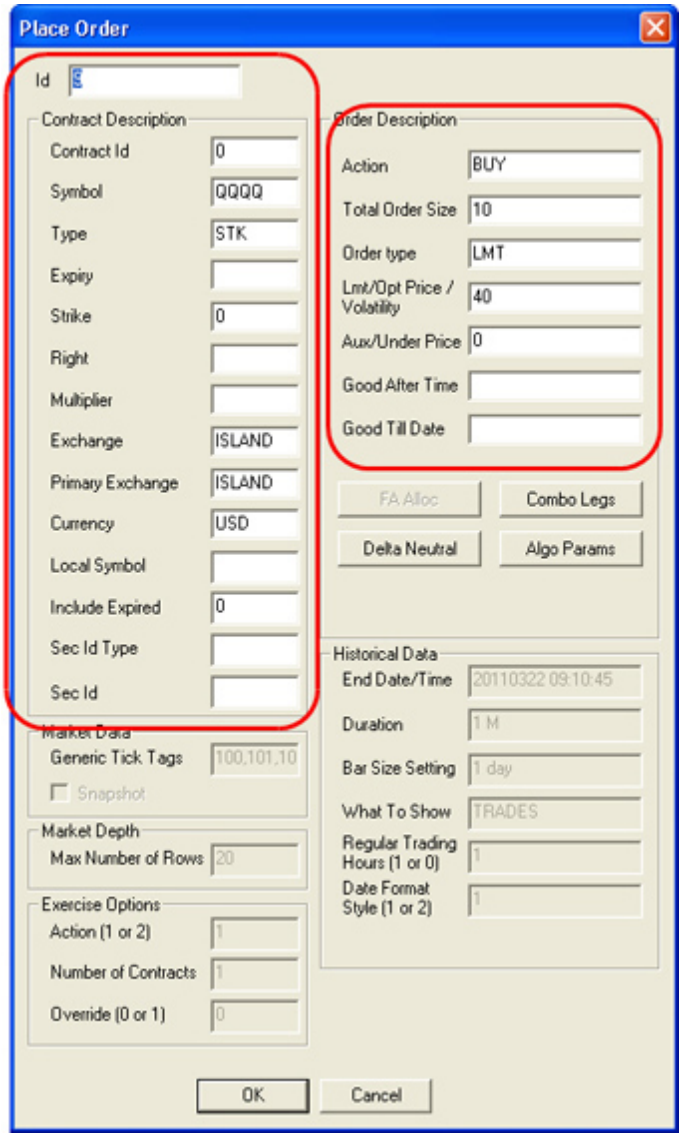
We've already seen several cases of On methods that call a corresponding EClientSocket method and display the appropriate dialog in the sample application. Placing orders is different. OnPlaceOrder() only calls the placeOrder() method, which in turn displays the Place Order dialog, and passes your contract and order information to TWS. Here is the placeOrder() method, defined in *client2Dlg.cpp*:

```
void CClient2Dlg::placeOrder(bool whatIf)
{
    // run order box
     m_dlgOrder->init(this, "Place Order", CDlgOrder::ORDER,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    Order& order = m_dlgOrder->getOrder();

    // save old and set new value of whatIf attribute
    bool savedWhatIf = order.whatIf;
    order.whatIf = whatIf;

    // place order
    m_pClient->placeOrder( m_dlgOrder->m_id, m_dlgOrder->getContract(),
       order);

     // restore whatIf attribute
     order.whatIf = savedWhatIf;
}
```

placeOrder() has three parameters: *id*, *contract* and *order*. *id* is the order Id. When the status of your order is returned from TWS, it will be identified by this Id. The *contract* parameter is the same EClientSocket property that contains all the attributes used to describe the requested contract, which in this case means all the values you entered in the Contract Description section of the Place Order dialog. The *order* parameter is another EClientSocket property that contains all the attributes used to describe your order, which correspond to the fields in the *Order Description* section of the Place Order dialog.

Parameter	Description
id	The order Id. You must specify a unique value. This tag is also used when canceling the order.
contract	This object contains attributes used to describe the contract.
order	This object contains attributes that describe the details of the order.

This table is for illustrative purposes only and is not intended to portray valid API documentation.



For a complete list of the properties in the *order* and *contract* structures, see the API Reference Guide.

C++ EWrapper Functions that Return Order Data

Once the order has been placed, and assuming there are no errors in the order, the C++ EWrapper function orderStatus() returns the current status of the order from TWS. The orderStatus() function and a list of all its parameters are shown below.

```
void CClient2Dlg::orderStatus( OrderId orderId, const CString &status, int
filled, int remaining, double avgFillPrice, int permId, int parentId,
double lastFillPrice, int clientId, const CString& whyHeld)
{
    // create string
    CString str;
    str.Format( "orderId=%i clientId=%i permId=%i status=%s  filled=%i
remaining=%i  avgFillPrice=%f lastFillPrice=%f parentId=%i whyHeld=%s",
       orderId, clientId, permId, (const char *)status, filled, remaining,
avgFillPrice, lastFillPrice, parentId, (const char*)whyHeld);

    // add to listbox
    int i = m_orderStatus.AddString( str);

    // move into view
    int top = i - N < 0 ? 0 : i - N;
    m_orderStatus.SetTopIndex( top);
}
```

Parameter	Description
id	The order ID that was specified previously in the call to placeOrder()
status	The order status. See The status Parameter later in this chapter for a list of all possible order statuses.
filled	Specifies the number of shares that have been executed.
remaining	Specifies the number of shares still outstanding.
avgFillPrice	The average price of the shares that have been executed. This parameter is valid only if the filled parameter value is greater than zero. Otherwise, the price parameter will be zero.
permId	The TWS id used to identify orders. Remains the same over TWS sessions.
parentId	The order ID of the parent order, used for bracket and auto trailing stop orders.
lastFilledPrice	The last price of the shares that have been executed. This parameter is valid only if the filled parameter value is greater than zero. Otherwise, the price parameter will be zero.
clientId	The ID of the client (or TWS) that placed the order. Note that TWS orders have a fixed clientId and orderId of 0 that distinguishes them from API orders.
whyHeld	This field is used to identify an order held when TWS is trying to locate shares for a short sell. The value used to indicate this is 'locate'.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

However, that's not all that happens when you place an order in the sample application. The status of your order, along with some other useful information, appears in the *TWS Server Responses* text panel. The additional information returned from TWS and displayed in the sample application includes:

- Open order information (if the order is not filled immediately)
- Details about the contract you ordered
- Details about the order
- List of extended attribute and their values as applied to your order
- Execution details (when the order is filled)
- Order state information

The screens on the following pages show all of the information that is returned from TWS and displayed in the *TWS Server Responses* text panel in response to a sample BUY LIMIT order for 100 shares of DELL stock.

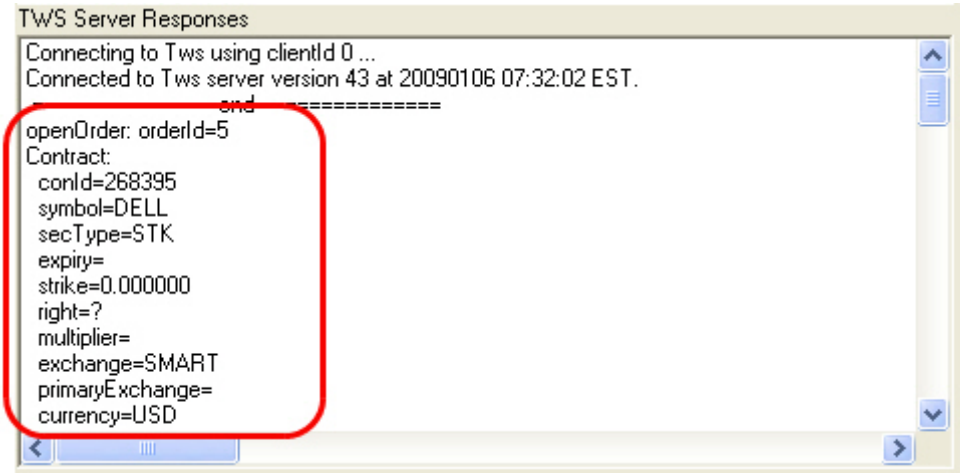
Open Order Information and Contract Details

If the order is not filled immediately, it is an open order. Open order information is returned by the EWrapper function openOrder(), about which we will learn more later in this chapter. For now, we're only interested in what it returns to the sample application, which is passed back through these parameters and shown in the screen below:

Parameter	Description
orderID	The order ID assigned by TWS. Use to cancel or update the order.
contract	This object includes attributes that describe the contract.
order	This object includes attributes that describe the details of the open order.
orderState	This object includes attributes that describe both pre and post trade margin and commission data.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

The order ID and the contract details of our fictional DELL order are displayed in the sample application as shown below.



And the order details are displayed as shown here:

For more information about the openOrder() function, see the API Reference Guide.

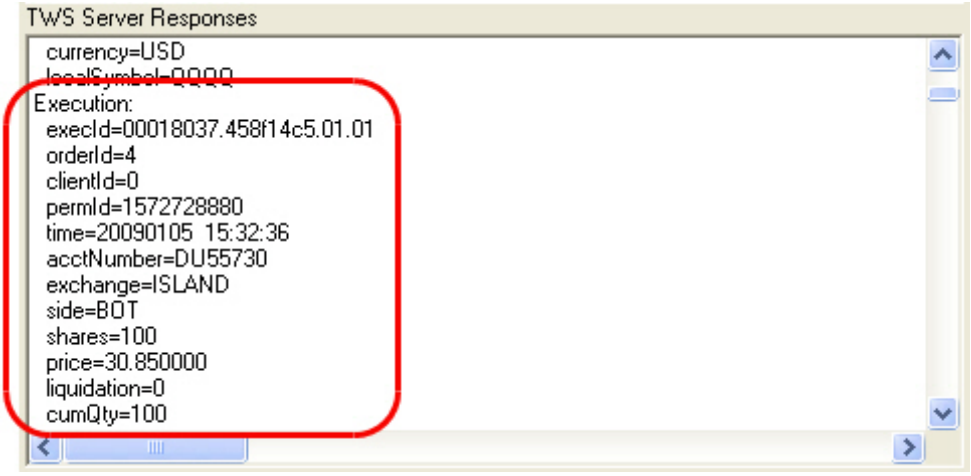
Extended Attributes

We'll learn more about extended attributes later in this chapter, but just remember that these attributes are part of the *order* EClientSocket property, which we just saw is returned wiht the openOrder() EWrapper function.

For more information about the *order* EClientSocket property, see the API Reference Guide.

Execution Details

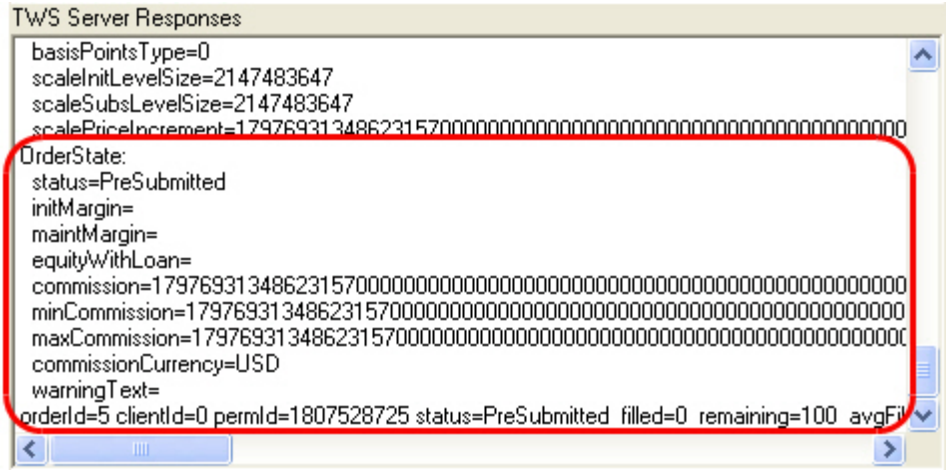
When the order is filled, execution details are sent from TWS in the EWrapper function execDetails() and displayed in the sample application as shown below.



We'll learn a bit more about executions later in this chapter.

Order State Information

Here is the order state information returned by the openOrder() function.



Note the last line in the screen; it contains your order status, and the data displayed on that line correspond to the parameters in the EWrapper function orderStatus().

The *status* Parameter

As we mentioned earlier, the *status* parameter in the orderStatus() function returns the status of your order and the sample application displays that information in the *TWS Server Responses* text panel. The possible values for the *status* parameter, and therefore, the possible order statuses, are:

- *PendingSubmit* - you have transmitted the order, but have not yet received confirmation that it has been accepted by the order destination.
- *PendingCancel* - you have sent a request to cancel the order but have not yet received cancel confirmation from the order destination. At this point, your order is not confirmed canceled. You may still receive an execution while your cancellation request is pending.

  Note: PendingSubmit and PendingCancel order statuses are not sent by the system and should be explicitly set by the API developer when an order is canceled.
- *PreSubmitted* - a simulated order type has been accepted by the system and that this order has yet to be elected. The order is held in the system until the election criteria are met. At that time the order is transmitted to the order destination as specified.
- *Submitted* - your order has been accepted at the order destination and is working.
- *Cancelled* - the balance of your order has been confirmed canceled by the system. This could occur unexpectedly when the destination has rejected your order.
- *Filled* - the order has been completely filled.
- *Inactive* - the order has been accepted by the system (simulated orders) or an exchange (native orders) but that currently the order is inactive due to sytem, exchange or other issues.
- *ApiPending* - the order has been reported to TWS by the API using reqAllOpenOrders() or reqOpenOrders().
- *ApiCancelled* - the order reported by the API has been cancelled.

Canceling an Order

To cancel an order, click the Cancel Order button on the main sample window, then click the OK button in the Place Order dialog. When you cancel your order, make sure the *Id* is the same as the orderID for your order (you can find this in the order status display in the *TWS Server Responses* text panel).

Let's take a look at the code behind this operation.

When you click the Cancel Order button, the OnCancelOrder() method defined in *client2Dlg.cpp* runs.

```
void CClient2Dlg::OnCancelOrder()
{
    // get order id
     m_dlgOrder->init( this, "Cancel Order", CDlgOrder::CANCEL_ORDER,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    // cancel order
    m_pClient->cancelOrder( m_dlgOrder->m_id);
}
```

The OnCancelOrder() method does the following:

- Displays the Cancel Order dialog (yes, this is yet another instance of our favorite dialog, *DlgOrder*.
- Calls the C++ method cancelOrder() when the OK button in the dialog is clicked.

The cancelOrder() Method

This method has a single parameter, id, which is the same order ID that was specified in the placeOrder() method. The cancelOrder() method is shown below.

```
m_pClient->cancelOrder( m_dlgOrder->m_id);
```

That's a lot of information about placing orders, but we're not done with orders yet! Turn the page to learn more about modifying orders and placing What-If orders.

Modifying an Order

To modify an order using the API, resubmit the order you want to modify using the same order id, but with the price or quantity modified as required. Only certain fields such as price or quantity can be altered using this method. If you want to change the order type or action, you will have to cancel the order and submit a new order.

Requesting "What-If" Data before You Place an Order

Another feature supported by the C++ sample application is the ability to request margin and commission "what if" data before you place an order. This means that you can click the What If button in the sample application, set up your order as if you were actually placing it, then see what the margins and commissions would be if the trade went through. The information is displayed in the *TWS Server Responses* text panel in the sample application.

Now we can go back and look at the methods in *client2Dlg.cpp* that we mentioned earlier. There is another On method called OnWhatIf() that does one thing: it calls the placeOrder() method but sets the boolean parameter *whatIf* to TRUE, which causes your order to NOT be placed and the margin and commission information to be displayed in the sample application.

Here are the lines of code that call the placeOrder() method in *client2Dlg.cpp*. To summarize: if you want to get what-if data, you click the What If button in the sample application, which sets the *whatIf* parameter to true. If you want to actually place your order, you click the Place Order button, which sets the *whatIf* parameter to true.

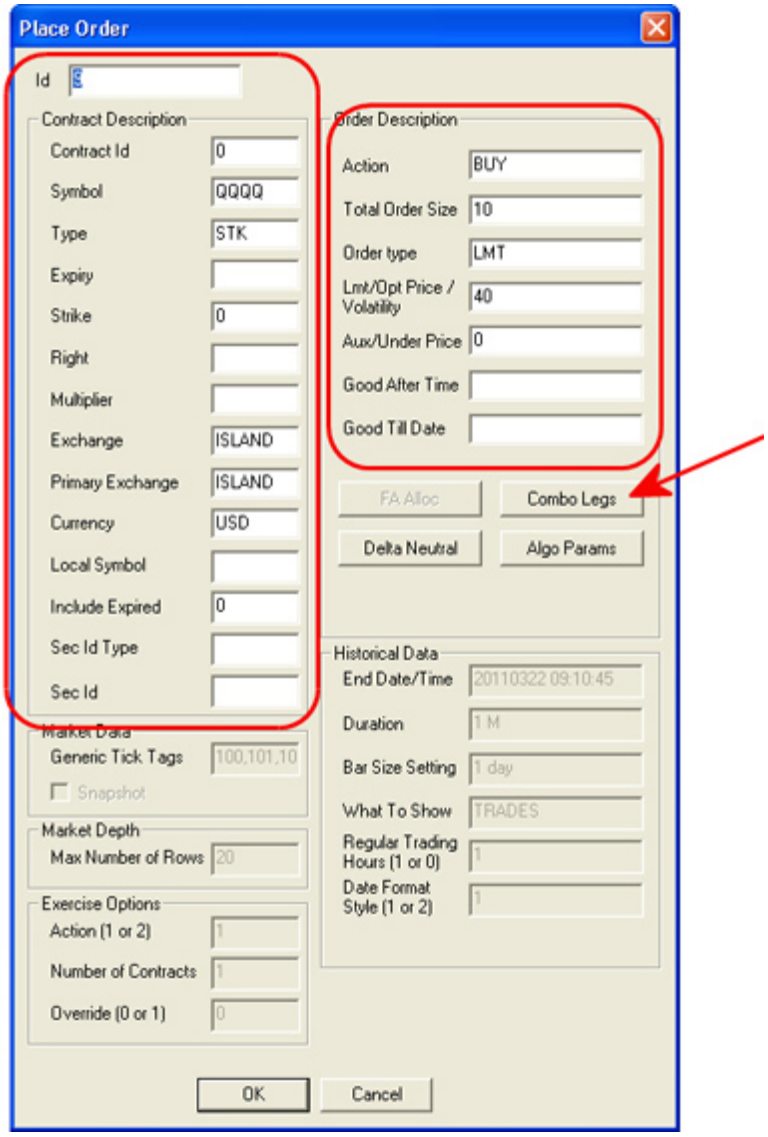
```
void CClient2Dlg::OnWhatIf()
{
    placeOrder(/* whatIf */ true);
}

void CClient2Dlg::OnPlaceOrder()
{
    placeOrder(/* whatIf */ false);
}
```

By the way, the *whatIf* parameter comes from our *Order* EClientSocket property, which if you recall from earlier in this chapter, is one of the parameters in the placeOrder() method.

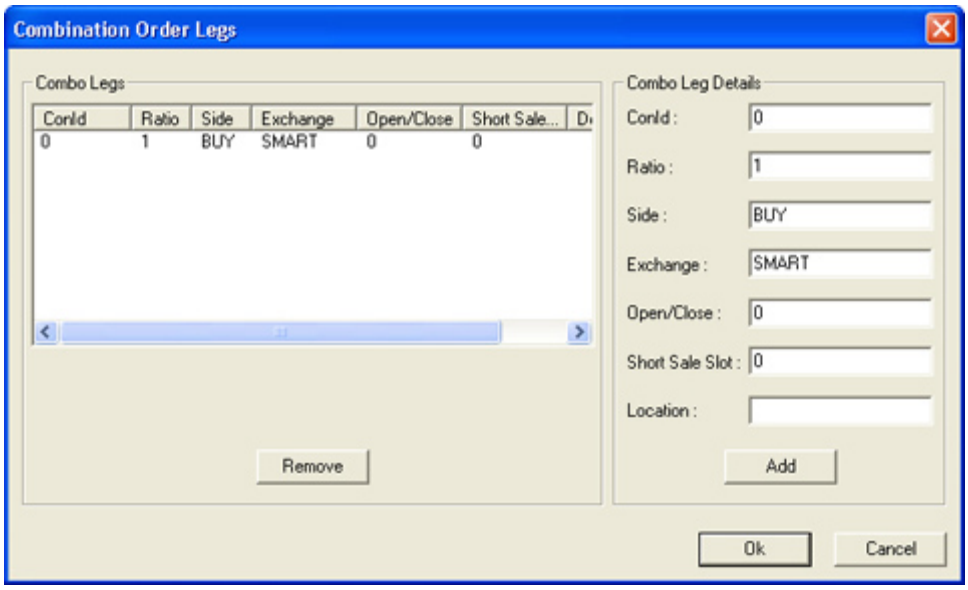
Placing Combo Orders

The TWS C++ API supports combination orders, which means that you can use the C++ API sample application to place combo orders that include options, stock and futures legs (or you can build your own application to place combo orders using the methods, events and parameters in the API).

To place a combo order in the sample application, simply place an order as you normally would by clicking the Place Order button, fill in the fields in the Ticker Description and Order Description sections of the Place Order dialog, then click the Combo Legs button as shown below, and add combo legs to the order in the Combination Order Legs dialog



And here is the Combination Order Legs dialog:



To add combo legs, simply fill in the fields in the *Combo Leg Details* section of the dialog, then click Add. Each leg you add appears in the list of combo legs on the left side of the dialog. You can remove unwanted legs by clicking the row to select it, then clicking the Remove button. When you click OK, you are returned to the Place Order dialog, where you click OK to place your order.

How is this process different in the code from the standard order placement? Let's find out!

Combo Legs Processing

As you might have expected, the first part of this process is the same as when you place a standard order: the OnPlaceOrder() method runs and does its thing, and placeOrder() method runs and does ITS thing, the Place Order dialog is displayed, and your input values are stored as parameters in placeOrder(). But when you click the Combo Legs button in the Place Order dialog, a different On method defined in *DlgOrder.cpp* (the Order dialog), OnBtnAddComboLegs() runs and displays the Combination Order Legs dialog. The actual combo leg list defined by you in the dialog is created by the code in the Combination Order Legs dialog (*DlgComboLegs.cpp*), then passed to TWS as a property of the *Contract* parameter in placeOrder(). Whew!

Here is what the OnBtnAddComboLegs() method looks like:

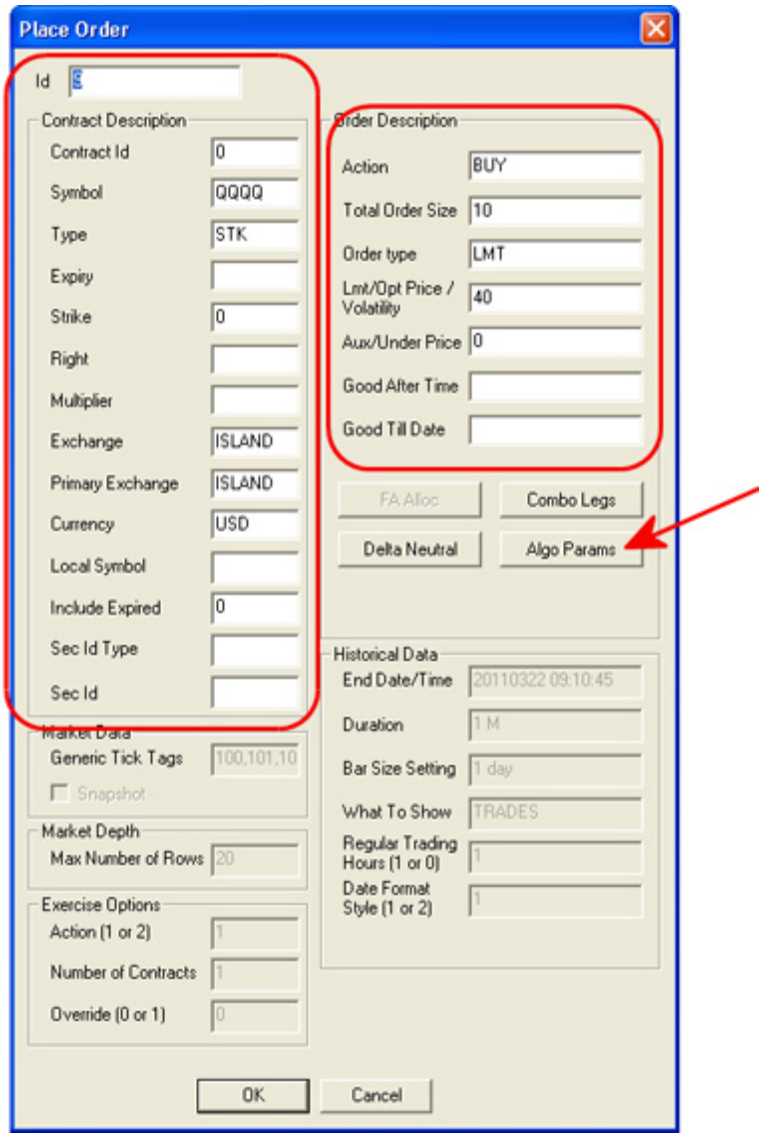
```
void CDlgOrder::OnBtnAddComboLegs()
{
    UpdateData();

    ComboLegList comboLegs;
    Contract::CloneComboLegs(comboLegs, m_comboLegs);

    CDlgComboLegs dlgComboLegs(comboLegs, m_exchange);
    if ( dlgComboLegs.DoModal() == IDOK )
       m_comboLegs.swap(comboLegs);

    Contract::CleanupComboLegs(comboLegs);
}
```

Just as in standard orders, once the order has been placed, and assuming there are no errors in the order, the C++ event orderStatus() returns the current status of the order from TWS, openOrder() returns information about open orders and execDetails() returns execution information.

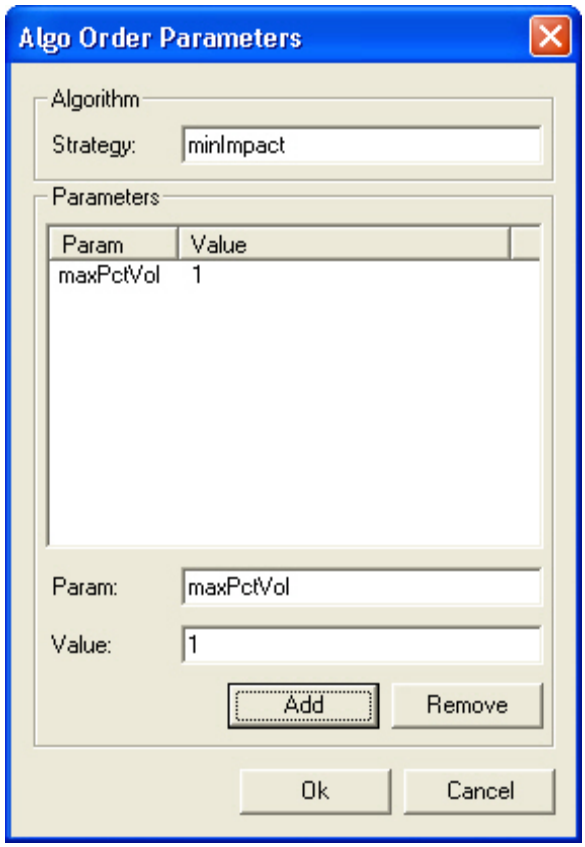
Placing Algo Orders

The TWS C++ API supports IBAlgo orders for US Equities and US Equity Options. Use IBAlgo orders to automatically balance market impact with risk on your large volume orders.

To place an IBAlgo order in the sample application, simply place an order as you normally would by clicking the Place Order button, fill in the fields in the *Ticker Description* and *Order Description* sections of the Place Order dialog, then click the Algo Params button as shown below and add an Algo strategy and Algo parameter/value pairs in the Algo Order Parameters dialog..



The Algo Order Parameters dialog appears:



To add Algo parameters to an order, type the name of the Algo strategy in the *Strategy field,* then type a parameter in the *Param* field, the value for the parameter in the *Value* field, and click Add. Repeat for each Algo parameter/value pair you want to add. For example, you might want to minimize market impact by slicing an options order over time as defined by the Max Percentage value. In this case you would type *minImpact* in the *Strategy* field, then type maxPctVol in the *Param* field and a percentage in the *Value* field. When you click OK, the Algo parameters are added to your order.

That's what happens on the user side of things; now let's take a look at the code.

Algo Order Processing

As you might have expected, the first part of this process is the same as when you place a standard order: the OnPlaceOrder() method runs and does its thing, and placeOrder() method runs and does ITS thing, the Place Order dialog is displayed, and your input values are stored as parameters in placeOrder(). But when you click the Algo Params button in the Place Order dialog, a different On method defined in *DlgOrder.cpp* (the Order dialog), OnBtnAlgoParams() runs and displays the Algo Order Parameters dialog. The actual Algo strategy and parameter/value list defined by you in the dialog is created by the code in the Algo Order Parameters dialog (*DlgAlgoParams.cpp*), then passed to TWS as properties of the *Order* parameter in placeOrder() (specifically, the *algoStrategy* and *algoParams* properties).

Here is what the OnBtnAlgoParams() method looks like:

```
void CDlgOrder::OnBtnAlgoParams()
{
    CDlgAlgoParams dlg(m_order->algoStrategy, m_order->algoParams);
    if ( dlg.DoModal() == IDOK ) {
        // do nothing - params passed by ref
        // and is being updated inside dialog
    }
}
```

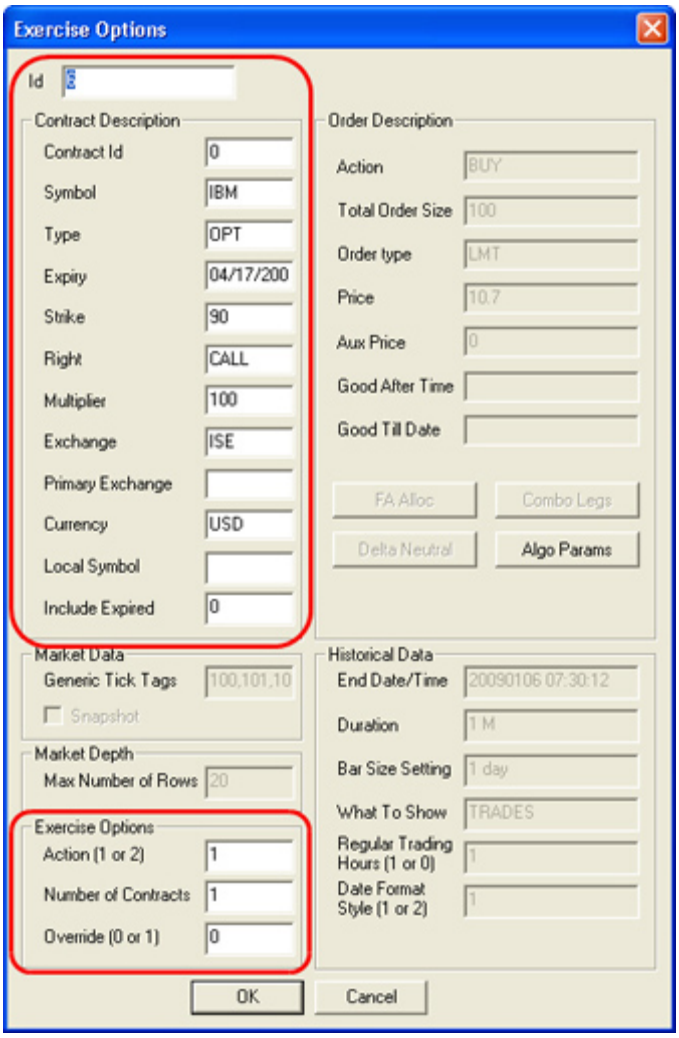
Just as in standard orders, once the order has been placed, and assuming there are no errors in the order, the C++ event orderStatus() returns the current status of the order from TWS, openOrder() returns information about open orders and execDetails() returns execution information.

You can also place Delta Neutral orders using the C++ API sample application. However, that type of order is beyond the scope of this guide. For more information, see the API Reference Guide.

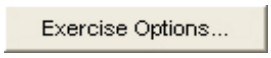
That's enough about orders. The next chapter describes what happens when you exercise options using the C++ API.

Chapter 14: Exercising Options

This chapter discusses how the C++ sample application exercises options prior to expiration, and instructs options to lapse. We'll also show you the methods, events and parameters behind the Options Exercise area of the sample application. The fields you complete in the Exercise Options dialog (another instance of the *dlgOrder* object, by the way) are shown below.



What Happens When I Click the Exercise Options Button?



When you click the Exercise Options button, the Exercise Options dialog appears. As we mentioned earlier, this is yet another version of the *dlgOrder* object. You enter the contract information in the *Contract Description* fields, then enter the option exercise information in the fields in the *Exercise Options* section (both sections are circled on the screen pictured on the previous page). In the *Action* field, enter *1* to exercise the option specified in the *Contract Description* fields, or *2* to let the option expire. Enter the number of contracts to either exercise or let expire in the *Number of Contracts* field. In the *Override* field, enter *1* to override the system's natural action, or *2* to not override. Click the OK button to execute your desired action (exercise or expire the option).

So what happens in the code when you do all this?

OnExerciseOptions()

When you click the Exercise Options button, the OnExerciseOptions() method defined in *client2Dlg.cpp* runs. Here is what the code for this On method looks like:

```
void CClient2Dlg::OnExerciseOptions()
{
    m_dlgOrder->init( this, "Exercise Options", CDlgOrder::EXERCISE_OPTIONS,
m_managedAccounts);
    if( m_dlgOrder->DoModal() != IDOK) return;

    m_pClient->exerciseOptions(
       m_dlgOrder->m_id,
       m_dlgOrder->getContract(),
       m_dlgOrder->m_exerciseAction,
       m_dlgOrder->m_exerciseQuantity,
       m_dlgOrder->getOrder().account,
       m_dlgOrder->m_exerciseOverride);
}
```

OnExerciseOptions() does the following:

- Initializes and displays the Exercise Options dialog.
- Calls the C++ method exerciseOptions() when the OK button is clicked.

The exerciseOptions() Method

When you click OK in the Exercise Options dialog, the exerciseOptions() method sends your request to TWS and, if all the entries are valid, your desired action is executed.

Now let's see which parameters are used when you exercise an option. The exerciseOptions() method looks like in our code:

```
m_pClient->exerciseOptions(
       m_dlgOrder->m_id,
       m_dlgOrder->getContract(),
       m_dlgOrder->m_exerciseAction,
       m_dlgOrder->m_exerciseQuantity,
       m_dlgOrder->getOrder().account,
       m_dlgOrder->m_exerciseOverride);
```

Let’s take a look at the parameters of this method.

Parameter	Description
tickerId	The Id for the option exercise request.
contract	This object includes attributes that describe the contract.
exerciseAction	This represents the action you want to take on the specified option. A value of *1* indicates that you want to exercise the option. *2* means that you want to let the option expire.
exerciseQuantity	The number of contracts to be exercised.
account	The IB account for institutional orders.
override	This parameter specifies whether your setting will override the system's natural action. For example, if your action is "exercise" and the option is not in-the-money, by natural action the option would not exercise. If you have override set to "yes" the natural action would be overridden and the out-of-the money option would be exercised. Values are *0* don’t override or *1* to override.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

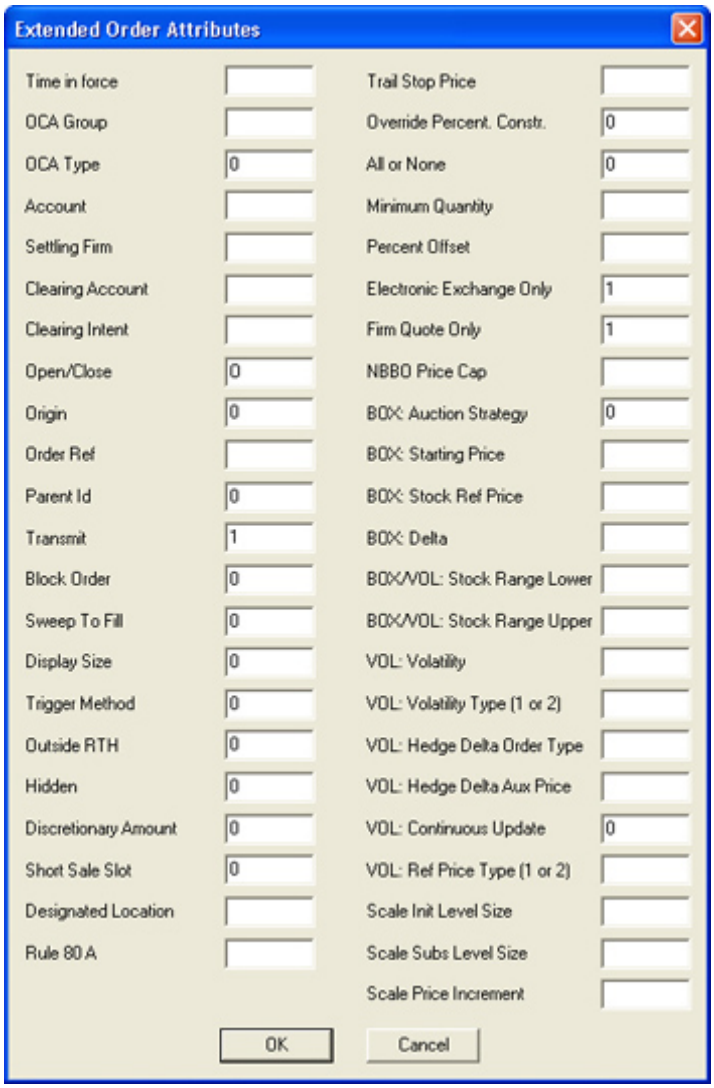
As you can see from the table above, this method has several parameters that correspond to the fields in the Exercise Options dialog, including our old friend *contract*, the EClientSocket property used throughout our API to describe a specific contract. This means that the values you entered in the dialog are passed to TWS by the parameters in the exerciseOptions() method.

In this case, there is no event that returns values from TWS.

Chapter 15: Extended Order Attributes

This chapter discusses how to apply extended, or non-essential, order attributes to your order. This sample action is different from many of the others we've looked at, as the extended order attributes for the C++ API are actually included in the *order* object, which you remember from our discussion on placing orders. For ease of use, the sample application has a separate dialog in which you can assign values to the extended order attributes. So although you will see a new dialog when you click the Extended button, the selections you're setting do not come from a new API method.

Here is the Extended Order Attributes dialog:



What Happens When I Click the Extended Button?

Extended order attributes have no function by themselves. However, any value you enter in the Extended Order Attributes dialog WILL be applied to every subsequent order you place in the sample application (until you either change or remove the value). In fact, you might remember from our earlier discussion of the types of data returned by TWS when you place an order that list of Extended Attributes displayed in the *TWS Server Responses* text panel. In case you forgot, here it is again:

You use these attributes to place advanced orders such as trailing stop limit, VOL and scale orders, as well modify other values for orders. Most important however is the fact that only those extended order attributes with values are applied to your order; the rest are ignored.

For a complete description of all of the extended order attributes in the C++ API, see the Extended Order Attributes topic in the API Reference Guide.

The code behind this process is fairly simple. When you click the Extended button, the OnExtord() method defined in *client2Dlg.cpp* runs. This method simply displays the Extended Order Attributes dialog, called *dlgExtOrd* in the code.

When you click the OK button in the Extended Order Attributes dialog, the values entered in the fields in the dialog are passed to the *order* object, which if you recall stores all the information about your order.

That's all the Extended button does. Until you place an order, the extended attributes are just that - attributes just sitting there waiting for something to happen. But once you create and place an order, the values you entered/modified in the Extended Order Attributes dialog are used in your order, and will continue to be applied to every order until you change them.

Next we'll take a look at some of the other buttons in the sample application.

Chapter 16: Requesting Open Orders

In this chapter, we're going to take a look at three related tasks in the C++ API sample application:

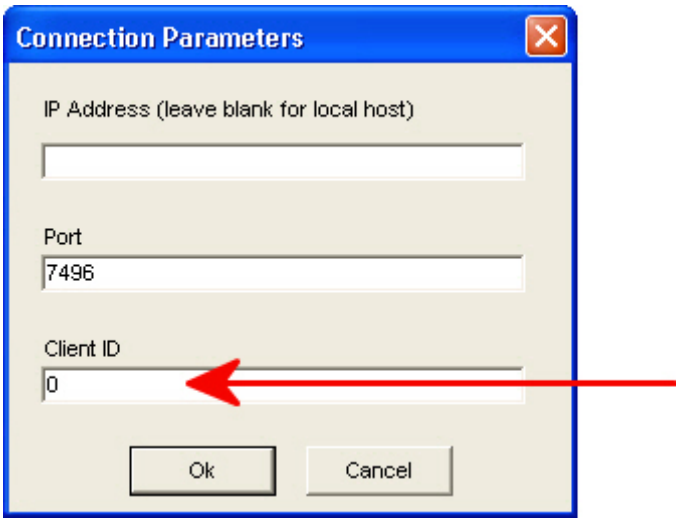
- Requesting Open Orders
- Requesting All Open Orders
- Requesting Auto Open Orders

How are they related?

Well, obviously they all give you information about open orders. The difference between them is the Client ID, which you set (or not!) when you connect to TWS.

Running Multiple API Sessions

You can connect up to eight API sessions to one TWS client, but the catch is that you have to assign a new client ID for each API session. Therefore, any orders sent from these clients can be tracked through the life of the order, and everyone knows where they came from and who's responsible for them. So be careful!:



If you happen to have TWS up and running now and want to try this out, simply run multiple sample API sessions as described in the following steps:

1 Click the Connect button and connect to the first session. Note that the *Client ID* is set to "0."

2 Do the same for another session. If you don't change anything, you'll see that you are not able to connect to this second session. In the *Errors and Messages* text panel on the sample application, the API will kindly tell you "Already connected."

3 Now try it with a unique Client ID. Click Connect again, only this time type 1 (or any other unique Client ID) in the *Client ID* field, then click OK.

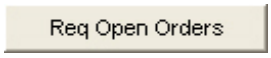
The Difference between the Three Request Open Orders Buttons

Now you're ready to learn the difference between the three Request Open Orders methods/buttons:

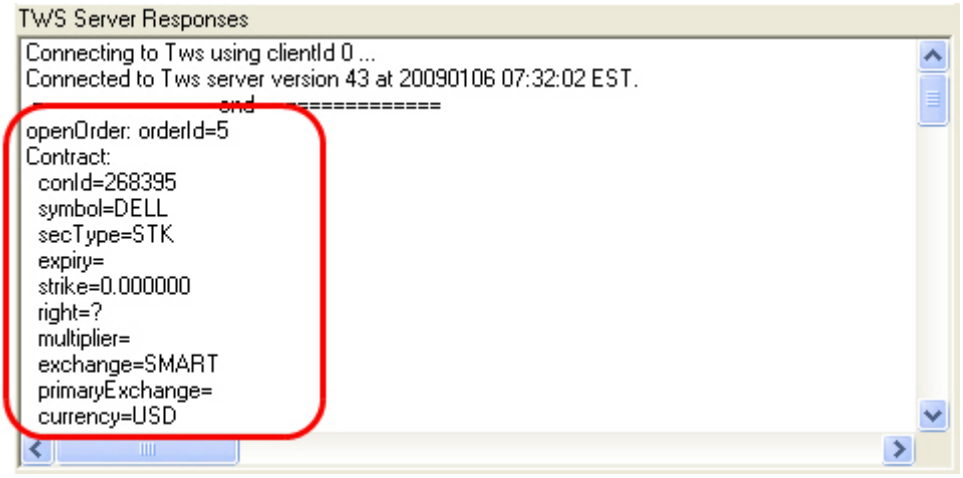
- Request Open Orders shows you any open orders made from that client, and if it's the "0" client ID client, you'll also see open orders sent from TWS.
- Request All Open Orders method shows you open orders sent from ALL clients connected to TWS, and all open orders that were sent from that TWS.
- Request Auto Open Orders method can only be used by the API with the client ID of "0." Clicking this button sets the boolean parameter to "True" and forever binds TWS orders to the API client. From that day forward, any time an open order exists on TWS it will automatically be returned via the Ewrapper methods, and in this case be displayed in the TWS Server Responses text panel of the sample application.

Got all that? Good, let's see the details.

What Happens When I Click the Req Open Orders Button?



When you click the Req Open Orders button, any open orders that currently exist are displayed in the *TWS Server Responses* text panel of the main sample application window, as shown below. You might recognize this screen; it's the same one we used to describe open order data returned from TWS as a result of an order.



If there are no open orders however, nothing will display in the panel.

Simple, right? So now let's see what happens in the code.

OnReqOpenOrders()

When you click the Req Open Orders button, the OnReqOpenOrders() method defined in *client2Dlg.cpp* calls the C++ method reqOpenOrders().

```
void CClient2Dlg::OnReqOpenOrders()
{
    // request open orders
    m_pClient->reqOpenOrders();
}
```

The reqOpenOrders() method, shown below, has no parameters.

```
m_pClient->reqOpenOrders();
```

C++ EWrapper Functions that Return Open Order Data

The reqOpenOrders() method triggers the EWrapper function openOrder(), which returns information about open orders and displays it in the *TWS Server Responses* text panel. The openOrder() method is shown below, followed by a list of its parameters.

```
void CClient2Dlg::openOrder( OrderId orderId, const Contract& contract,
const Order& order, const OrderState& orderState)
```

Parameter	Description
orderID	The order ID assigned by TWS. Use to cancel or update the order.
contract	This object includes attributes that describe the contract.
order	This object includes attributes that describe the details of the open order.
orderState	This object includes attributes used for both pre and post trade margin and commission data.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

If you've read this book from the beginning, by now you should be familiar with the *contract* and *order* EClientSocket properties, which contain properties that represent, respectively, a contract and an order. The openOrder() function receives open order information via these two structures, and also receives information from the *OrderState* object, which contains properties representing the margin and commissions fields for both pre- and post-trade data.

There is one additional open order function used in the C++ API,openOrderEnd(). This function, which has no parameters, serves as an end marker for a set of received open orders. openOrderEnd() is called when all orders are sent to a client as a response to the reqOpenOrders() method.

The code for openOrderEnd() displays the end marker for the open orders information displayed in the *TWS Server Responses* text panel of the sample application main window.

What Happens When I Click the Req All Open Orders Button?

When you click the Req All Open Orders button, open orders sent from ALL clients connected to TWS, and all open orders that were sent from this client are displayed in the *TWS Server Responses* text panel of the main sample application window, as shown below.

Let's see what happens in the code.

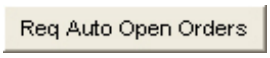
OnReqAllOpenOrders()

When you click the Req All Open Orders button, the OnReqAllOpenOrders() method defined in *client2Dlg.cpp* calls the C++ method reqAllOpenOrders().

```
void CClient2Dlg::OnReqAllOpenOrders()
{
    // request list of all open orders
    m_pClient->reqAllOpenOrders();
}
```

The reqAllOpenOrders() method has no parameters but it does trigger the EWrapper function openOrder(), which returns information about open orders and displays it in the *TWS Server Responses* text panel, just as it does in response to the reqOpenOrders() method. See the section on C++ EWrapper Functions that Return Order Data earlier in this chapter for details about this function.

What Happens When I Click the Req Auto Open Orders Button?



When you click the Req All Open Orders button, TWS orders are bound to the API client (the client you are running!). From that day forward, any time an open order exists on TWS it will automatically be returned via C++ EWrapper function openOrder() and displayed in the *TWS Server Responses* text panel of the sample application.

Note that this function can only be used by the API with the client ID of "0."

Let's see what happens in the code.

OnReqAutoOpenOrders()

When you click the Req All Open Orders button, the OnReqAutoOpenOrders() method defined in *client2Dlg.cpp* calls the C++ EClient Socket method reqAutoOpenOrders().

```
void CClient2Dlg::OnReqAutoOpenOrders()
{
    // request to automatically bind any newly entered TWS orders
    // to this API client. NOTE: TWS orders can only be bound to
    // client's with clientId=0.
    m_pClient->reqAutoOpenOrders( true);
}
```

The reqAutoOpenOrders() Method

This method has a single parameter, *bAutoBind*. If this parameter is set to *true* (and notice *(true)* in the call to the reqAutoOpenOrders() method above), newly created orders will be implicitly associated with the client making the Auto Open Orders request. If the parameter is set to *false*, no association is made.

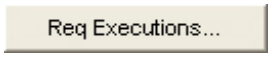
In other words, if you are using an API with a Client ID of "0," the *autoBind* parameter in reqAutoOpenOrders is set to true and orders from ALL clients connected to TWS will be reported to the sample application. If you're not Client ID 0, you'll receive an error message and the auto-binding won't be enabled.

The reqAutoOpenOrders() method triggers the openOrder() event, which returns information about open orders and displays it in the *TWS Server Responses* text panel, just as it does in response to the reqOpenOrders() method. See the section on C++ EWrapper Functions that Return Order Data earlier in this chapter for details about this function.

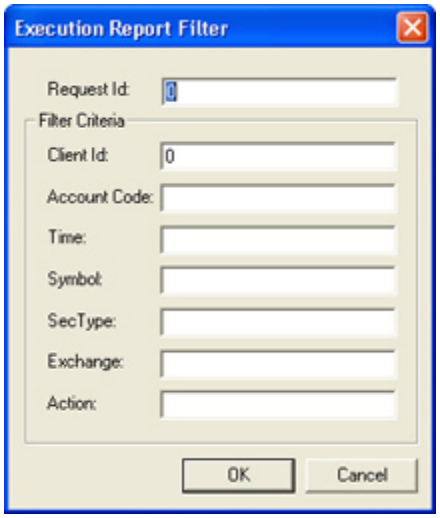
Chapter 17: Requesting Executions

This chapter shows you how to request execution reports using the C++ sample application, and how our API handles such requests. You can retrieve all execution reports, or only those you want by entering specific criteria such as time, symbol, exchange and more. We'll show you how to use the sample application to get these execution reports, and we'll see the methods, events and parameters behind the process.

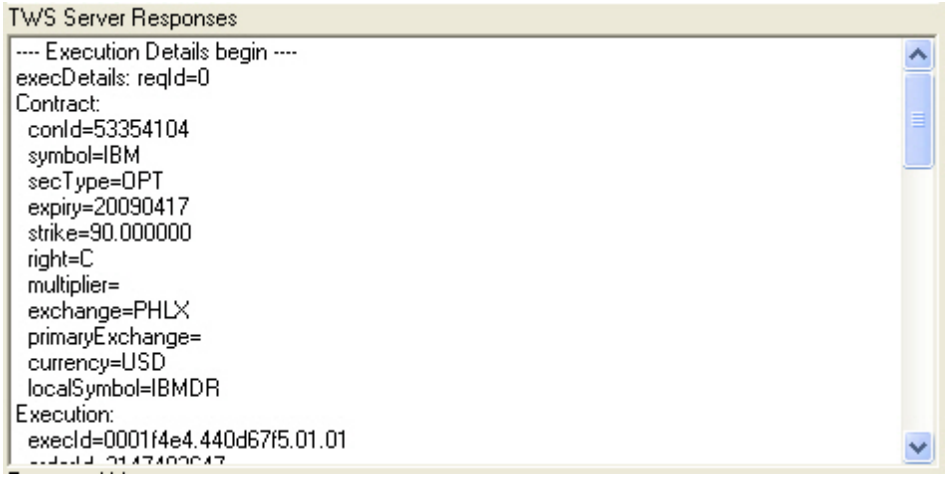
What Happens When I Click the Req Executions Button?



When you click the Req Executions button, the Execution Report Filter dialog appears. You enter filter criteria for your execution reports by filling in the fields. You can filter your execution reports by request ID, client ID, account code, time, symbol, security type, exchange or action. If you leave all the fields blank except the *Client ID* field (which is filled in with "0" by default), you will get reports of all of your executions.



The report is displayed in the *TWS Server Responses* text panel:



Now let's take a look at the code.

OnReqExecutions()

When you click the Req Executions button, the OnReqExecutions() method defined in *client2Dlg.cpp* runs. Here is what the code for this method looks like:

```
void CClient2Dlg::OnReqExecutions()
{
    CRptFilterDlg dlg(m_execFilter.get());

    if ( dlg.DoModal() != IDOK) return;

   m_pClient->reqExecutions(dlg.reqId(), *m_execFilter);
}
```

OnReqExecutions() does the following:

- Initializes and displays the Execution Report Filter dialog, *RptFilterDlg*.
- Calls the C++ method reqExecutions().

We mentioned before that the *Client ID* field comes with a default value of "0." This isn't by chance! You can leave all of the other fields blank and everything will be fine. But if you leave the *Client ID* field blank, you'll get nothing, no matter what other field values you may enter. After you define the filter criteria and click OK, your values are passed to TWS via the reqExecutions() method.

The reqExecutions() Method

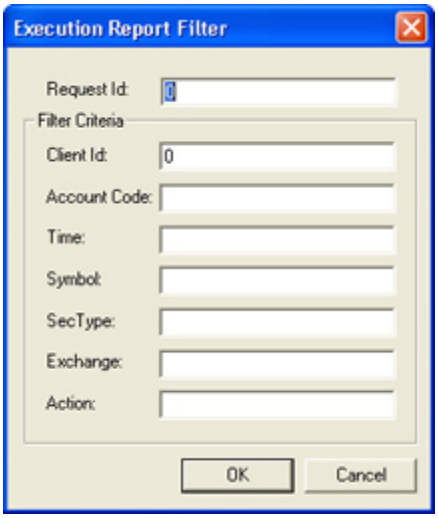
The reqExecutions() method sends the values you entered in the Execution Report Filter dialog to TWS. Another way of saying this is that the filter criteria you entered in the Execution Filter dialog are the parameters for this method, which is shown below along with its parameters.

```
m_pClient->reqExecutions(dlg.reqId(), *m_execFilter);
```

Parameter	Description
reqId	The ID of the data request. Ensures that responses are matched to requests if several requests are in process.
filter	This object contains attributes that describe the filter criteria used to determine which execution reports are returned.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

The *filter* parameter is another one of our EClientSocket properties, *ExecutionFilter*. It contains attributes such time, symbol, security type, exchange, and other execution report filter criteria. These attributes correspond to the fields in the Execution Report Filter dialog in our sample application.



For a complete list of the properties in the *ExecutionFilter* COM object, see the API Reference Guide.

That concludes our discussion of how orders and executions are handled by our C++ API. The next section of the guide introduces you to some additional trading tasks supported by our API and the sample application.

C++ EWrapper Functions that Return Execution Details

Execution reports are returned via the execDetails() EWrapper function.

```
void CClient2Dlg::execDetails( int reqId, const Contract& contract, const
Execution& execution)
{
    int i = m_orderStatus.AddString("---- Execution Details begin ----");

    // create string
    CString str;
    str.Format("execDetails: reqId=%i", reqId);

    // add to listbox
    i =  m_orderStatus.AddString(str);
.
.
.
```

The code for this method is much longer than what's shown above (the missing portion contains code for displaying the execution details in the sample applictation). As you can see from the event, execDetails() contains the following parameters:

Parameter	Description
orderId	The order Id that was specified previously in the call to placeOrder().
contract	This object includes attributes that describe the contract.
execution	This object includes attributes that describe additional execution details.
reqId	The ID of the data request. Ensures that responses are matched to requests if several requests are in process.

Tables are for illustrative purposes only and are not intended to represent valid API information.

The detailed information about your executions are included as properties of the *execution* object, which is one of our EClientSocket properties. The *contract* object contains information about the contract that was traded; yes, this is the same EClientSocket property you've seen before. For a complete list of the properties in the *execution* and *contract* objects, see the API Reference Guide.

execDetailEnd()

There is one additional Ewrapper function involved in getting execution reports from TWS: the execDetailsEnd() event. This function, which has a single parameter (*reqId)*, serves as an end marker for a set of received execution reports. It is called when all executions have been sent to a client as a response to the reqExecutions() method. Here is what the execDetailsEnd() event looks like:

```
void CClient2Dlg::execDetailsEnd( int reqId)
{
    CString str;
    str.Format("reqId=%i =============== end ===============", reqId);

    int i = m_orderStatus.AddString(str);

    // bring into view
    int top = i - N < 0 ? 0 : i - N;
    m_orderStatus.SetTopIndex(top);
}
```

execDetailsEnd() also displays the end marker for the execution reports displayed in the TWS Server Responses text panel of the sample application main window.

This concludes the section on orders and order information. The next section discusses the remaining tasks that you can perform using the C++ sample application (or using your own custom application!), including requesting the current server time and subscribing to news bulletins.

6

Additional Tasks

This section describes some additional tasks that you can perform using the C++ API sample application. These operations don’t really belong anywhere else in this guide, so we’ve grouped them together here. We'll show you the methods and parameters behind such tasks as requesting the current server time, the next ID, subscribing and unsubscribing to news bulletins, and changing the server logging level.

Here's what you'll find in this section:

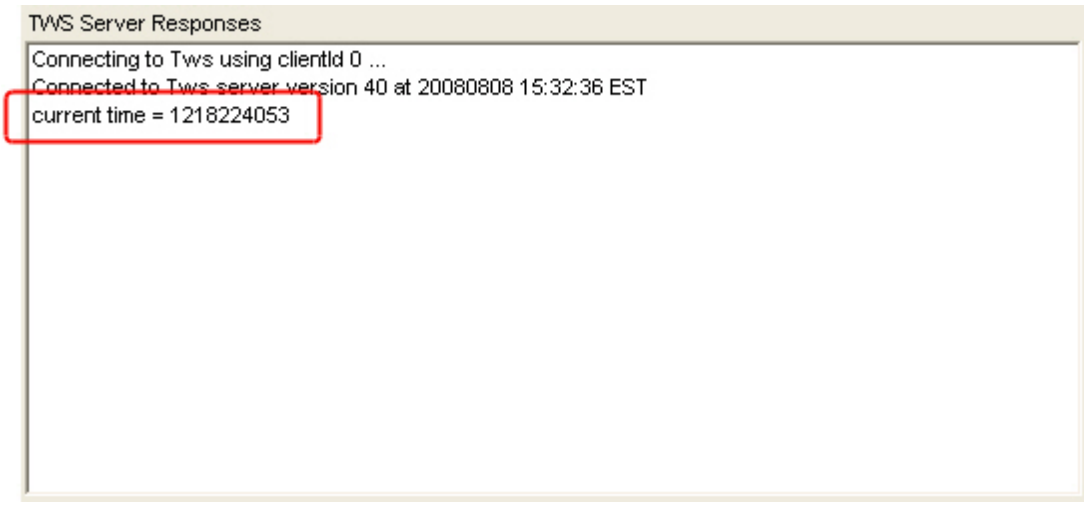
In addition to the tasks described in this chapter, the ActiveX API sample application also includes a few more advanced functions, including the ability to calculate volatility and option price, and support for IBAlgos. For more information on these and other advanced capabilities of the ActiveX API, see our API Reference Guide, available from the Reference Guide tab on our IB API web page.

Chapter 18 - Requesting the Current Time

This chapter discusses the method for requesting the current server time. Actually, "discusses" is really not the correct word. It merely "states" the method, which is quite solitary with no parameters to call its own.

What Happens When I Click the Current Time Button?

You request the current server time by clicking the Current Time button The server time is displayed in the *TWS Server Responses* text panel, as shown below.



This is a very simple process from a user's point of view and the code that makes this happen is also quite simple.

OnReqCurrentTime()

Just like the other buttons in the sample application, the Current Time button has an On method associated with it. When you click the button, the OnReqCurrentTime() method defined in *client2Dlg.cpp* runs. Here is what the code for this method looks like:

```
void CClient2Dlg::OnReqCurrentTime()
{
    m_pClient->reqCurrentTime();
}
```

OnReqCurrentTime() calls the C++ EClient Socket method reqCurrentTime(), and that's pretty much all it does!

The reqCurrentTime() Method

The reqCurrentTime() method is so simple that is doesn't even have any parameters. All it does is trigger the currentTime() event, which returns the current server time.

```
m_pClient->reqCurrentTime();
```

C++ EWrapper Functions that Return the Current Time

There is one EWrapper function that returns the current time, and that function is currentTime(). This function has only one parameter, *time*, which as you might have guessed, returns the current time on the server.

The currentTime() function is shown below.

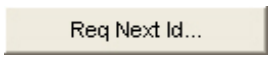
```
void CClient2Dlg::currentTime(long time)
{
    CString displayString;
    displayString.Format("current time = %d", time);
    int i =  m_orderStatus.AddString(displayString);

    // bring into view
    int top = i - N < 0 ? 0 : i - N;
    m_orderStatus.SetTopIndex( top);
}
```

Chapter 19: Requesting the Next Order ID

Each order you place in TWS (and in an API application) has a unique order ID assigned to it. There is a rule about order IDs in TWS: Each successive order ID must be greater than the most recently used order ID. As an example, consider the situation in which you place an orders using order ID 1, then place an order using order ID 5. The next available order ID would be 6; you can never go back and use 2, 3 or 4. Order ID 6 is greater than order ID 5, the most recently used order ID.

You can use the TWS C++ API to request the next valid order ID that can be used when placing an order. You might use this functionality if you are creating your own custom trading application and with to ensure that each order uses a legal order ID.

What Happens When I Click the Req Next Id Button?



OnReqIds()

When you click the button, the OnReqIds() method defined in *client2Dlg.cpp* runs. Here is what the code for this method looks like:

```
void CClient2Dlg::OnReqIds()
{
    // request a block of 20 id's;
    // next id is returned via nextValidId()
    m_pClient->reqIds( 20);
}
```

This method does one thing: it calls the EClient Socket method reqIds().

The reqIds() Method

The C++ method that you use to request the next valid ID is reqIDs(). After calling this method, the nextValidId() EWrapper function is triggered, and the next valid ID is returned from TWS. That ID will reflect any autobinding that has occurred (which generates new IDs and increments the next valid ID therein).

The reqIDs() method is shown below.

```
m_pClient->reqIds( 20);
```

reqIDs() has a single parameter, *numIds*. This parameter however is simply a placeholder and has no real purpose. Simply set this parameter to any integer to make the method work the way it's supposed to. In the example shown above, this parameter is set to 20.

C++ EWrapper Functions that Return the Next Valid Id

As we mentioned above, the next valid order ID is returned from TWS via the nextValidId() function. This function is also triggered when you successfully connect to TWS. The nextValidId() event looks like this:

```
void CClient2Dlg::nextValidId( OrderId orderId)
{
    m_dlgOrder->m_id = orderId;
}
```

There is only one parameter returned with this event, *orderId*, which as you probably figured out by now returns the next available order ID received from TWS. Increment all successive orders by one based on this ID.

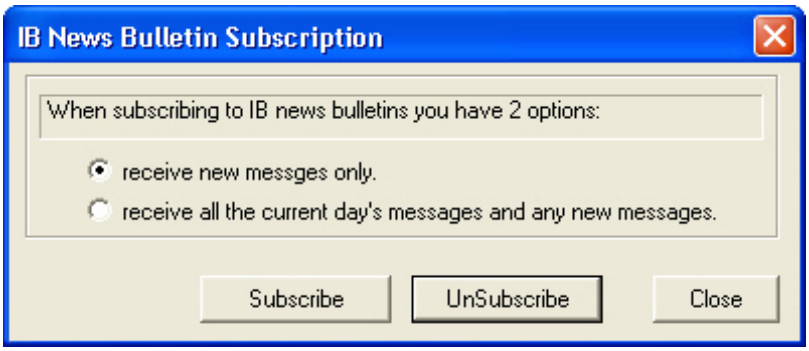
Chapter 20: Subscribing to News Bulletins

This chapter shows you how to subscribe to IB news bulletins through the C++ sample application. Once you subscribe, all bulletins will display in the *TWS Server Responses* text panel of the sample application. The news bulletins keep you informed of important exchange disruptions.

We will show you the methods, events and parameters responsible for letting you subscribe and unsubscribe to news bulletin feature in the C++ sample application.

What Happens When I Click the Req News Bulletins Button?

When you click the Req News Bulletins button, the IB News Bulletin Subscription dialog appears. In the dialog, you can elect to receive new messages only, or receive all the current day's messages and any new messages. These two options are presented as radio buttons. After you select your choice, click the Subscribe button to submit your subscription.



That's how you subscribe to news bulletins using the C++ API sample application. Keep reading to learn what happens in the code during this process.

OnNewsBulletins()

When you click the Req News Bulletins button, the OnNewsBulletins() method defined in *client2Dlg.cpp* runs. Here is what the code looks like:

```
void CClient2Dlg::OnNewsBulletins()
{
    CDlgNewsBulletins dlg;

    if (dlg.DoModal() != IDOK) return;

    dlg.subscribe()
        ? m_pClient->reqNewsBulletins( dlg.allMsgs())
        : m_pClient->cancelNewsBulletins();
}
```

OnNewsBulletins() does the following:

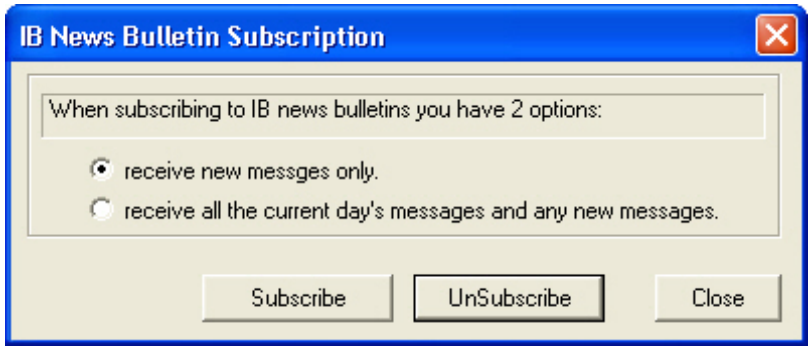
- Initializes and displays the IB News Bulletin Subscription dialog, dlgNewsBulletins.
- Calls the C++ method reqNewsBulletins() if the user clicks the Subscribe button in the IB News Bulletin Subscription dialog.
- Calls the C++ method cancelNewsBulletins() if the user clicks the UnSubscribe button in the IB News Bulletin Subscription dialog.

The reqNewsBulletins() method

This method tells TWS that you want to subscribe to news bulletins.

```
m_pClient->reqNewsBulletins( dlg.allMsgs())
```

reqNewsBulletins() has one parameter: *allMsgs*. If you select the *receive new messages only* radio button in the IB News Bulletin Subscription dialog, the *allMsgs* parameter, which asks "receive ALL messages, old and new?" will be set to false, which basically means that you will receive only new news bulletins. If you select *receive all the current day's messages and any new messages*, the *allMsgs* parameter is set to true, which means, that you will receive all news bulletins for the current day PLUS any new news bulletins. Either way, you are now subscribed to news bulletins, and either way you will receive any NEW bulletins that get posted fromthe time you subscribe.



News bulletins are returned to the C++ sample application via the updateNewsBulletin() event.

C++ EWrapper Functions that Return News Bulletins

The bulletins are returned via the updateNewsBulletin() EWrapper function.

```
void CClient2Dlg::updateNewsBulletin(int msgId, int msgType, const
CString& newsMessage, const CString& originExch)
{
    CString displayString;
   displayString.Format(" MsgId=%d :: MsgType = %d :: Origin= %s ::
Message= %s",
            msgId, msgType, originExch, newsMessage);

    MessageBox( displayString, "IB News Bulletin", MB_ICONINFORMATION);
}
```

updateNewsBulletin() contains the following parameters:

Parameter	Description
msgId	The bulletin ID, incrementing for each new bulletin.
msgType	Specifies the type of bulletin. Valid values include: • 1 = Requlars news bulletin • 2 = Exchange no longer available for trading • 3 = Exchange is available for trading
message	The bulletin's message text.
origExchange	The exchange from which this message originated.

Tables are for illustrative purposes only and are not intended to represent valid API information.

Canceling News Bulletins

If you're tired of knowing what's going on around you, you can elect to unsubscribe, or cancel the news bulletins. To unsubscribe to news bulletin, you first need to click the Req News Bulletins button in the C++ sample application. Then click Unsubscribe in the IB News Bulletin Subscription dialog. When you do this, we call the cancelNewsBulletins() method, which as the name implies, cancels your news bulletin subscription.

The cancelNewsBulletins() method header looks like this:

```
m_pClient->cancelNewsBulletins();
```

Because you are simply canceling a request, there are no values returned by this method.

Chapter 21: Viewing and Changing the Server Logging Level

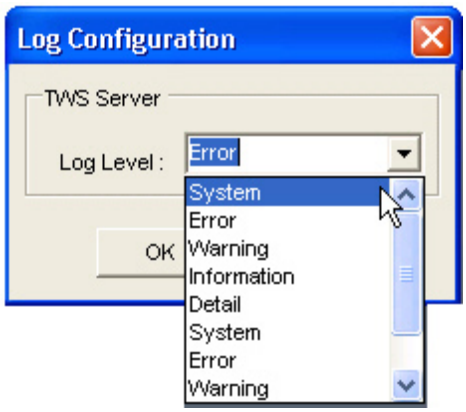
This chapter shows you how to view and change the server logging level.

As client requests are processed (both system and API clients), TWS logs certain information to its log.txt log file located in the installation directory. The purpose of this file is to help resolve problems by providing some insight into the state of the program before the problem occurred. In the C++ sample application, you can specify how detailed the information will be when entered into the log.txt file. Basically, the higher the log level, the more performance overhead that may be incurred. By default, the server logging level is set to "2" for error logging.

See our API Reference Guide for more information about API logging. The API Reference Guide is available from the Application Programming Interfaces page on our web site as an online guide or a downloadable/printable PDF.

What Happens When I Click the Log Configuration Button?

To see or change the server logging level, you first click the Log Configuration button on the C++ sample application. In the Log Configuration dialog that appears, you select the logging level from the drop-down. You can select *System*, *Error*, *Warning*, *Information* or *Detail*. After you make your selection, click OK to close the dialog. Of course, you won't really see any changes in the sample application unless you encounter a problem of some kind.



That's what happens on the user side of things. Let's see what happens in the code.

OnSetServerLogLevel()

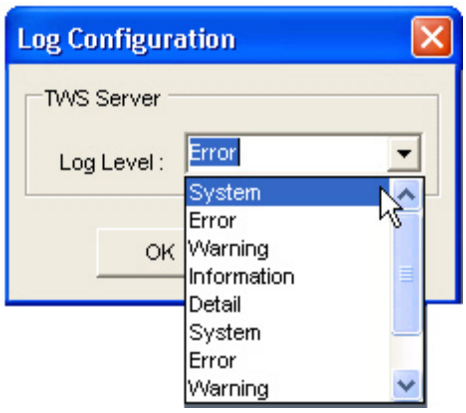
As with all the other buttons on the sample applications, when you click the Log Configuration button, an On method in *client2Dlg.cpp* runs. The On method for this button is called OnSetServerLogLevel() and is shown below.

```
void CClient2Dlg::OnSetServerLogLevel()
{
    if (s_dlgLogConfig.DoModal() != IDOK) return;

    // set the TWS log level for API requests/responses
    m_pClient->setServerLogLevel( s_dlgLogConfig.serverLogLevel() );
}
```

OnSetServerLogLevel() does the following:

- Initializes and displays the Log Configuration dialog (the *dlgLogConfig* object).
- Calls the C++ method setServerLogLevel() when the OK button is clicked.

The default level appears in the *Log Level* field of the Log Configuration dialog. We've expanded the dropdown list in the following figure just to show you the available log levels. Once you select a level and click OK, we call the setServerLogLevel() method.



The setServerLogLevel() Method

The setServerLogLevel() method contains a single parameter, *logLevel*. This parameter passes the log level you selected in the Log Configuration dialog to TWS.

```
m_pClient->setServerLogLevel( s_dlgLogConfig.serverLogLevel() );
```

The *logLevel* parameter specifies the level of log entry detail used by TWS when processing API requests. The valid values for this parameter correspond to the choices in the Log Level dropdown in the Log Configuration dialog:

- 1 = SYSTEM
- 2 = ERROR
- 3 = WARNING
- 4 = INFORMATION
- 5 = DETAIL

For more information about log levels and log entries, see the API Logging topic in the API Reference Guide.

There are no parameters passed from TWS to the API in this process; therefore, there is no corresponding C++ EWrapper function.

This concludes our discussion of the C++ sample application for individual accounts. The next section discusses how the C++ API sample application takes care or Financial Advisors and multi-client accounts.

7

Where To Go From Here

If you've come this far and actually read the book, you now have a pretty decent grasp on what the C++ API can do, and how to make it do some of the things you want. Now we give you a bit more information about how to link to TWS with our C++ API, and we suggest some helpful outside resources you can use to help you move forward.

This section contains the following chapters:

Chapter 22 - Linking to TWS using the TWS C++ API

If you have the skill and confidence to handle C++ on your own, you can build your own C++ API application to link to TWS using the following steps as a guide.

To link to TWS using the TwsSocketClient.dll

1 Create a Windows application using MS Visual Studio (version 5.0 or higher).

2 Add C:`\jts\SocketClient\include` to your project's include path. This should be done for any individual project that accesses the TwsSocketClient library's header files.

3 Add the C:`\jts\SocketClient\lib\TwsSocketclient.lib` file to your project's libraries path. This should be done for any individual project that accesses the TwsSocketClient library.

4 Include EWrapper.h and EClientSocket.h in any Visual C++ source code that accesses their functionality and data structures.

5 Subclass the EWrapper class.

6 Override the following functions:

Ewrapper Function	Description
tickPrice()	Handles market data.
tickSize()]	
tickOptionComputation()	
tickGeneric()	
tickString()	
tickEFP()	
orderStatus()	Receives order status.
openOrder()	Receives open orders.
error()	Receives error information.
connectionClosed()	Notifies when TWS terminates the connection.
updateAccountValue()	Receives current account values.
updateAccountTime()	Receives the last time account information was updated.
updatePortfolio()	Receives current portfolio information.
nextValidId()	Receives the next valid order ID upon connection.
contractDetails()	Receives contract information.
contractDetailsEnd()	Identifies the end of a given contract details request.

Ewrapper Function	Description
bondContractDetails()	Receives bond contract information.
exectDetails()	Receives execution report information.
updateMktDepth()	Receives market depth information.
updateMktDepthL2()	Receives Level II market depth information.
updateNewsBulletin()	Receives IB news bulletins.
managedAccounts()	Receives a list of Financial Advisor (FA) managed accounts.
receiveFA()	Receives FA configuration information.
historicalData()	Receives historical data results.
scannerParameters()	Receives an XML document that describes the valid parameters of a scanner subscription.
scannerData()	Receives market scanner results.
realTimeBar()	Receives real-time bars.
currentTime()	Receives the current system time on the server.
fundamentalData()	Receives Reuters global fundamental market data.

7 Instantiate the EClientSocket class.

8 Call the following functions:

a Import com.ib.client.* into your source code file.

b Implement the EWrapper interface. This class will receive messages from the socket.

c Override the following functions:

Ewrapper Functions	Description
tickPrice()	Handles market data.
tickSize()	
tickOptionComputation()	
tickGeneric()	
tickString()	
tickEFP()	
orderStatus()	Receives order status.
openOrder()	Receives open orders.
error()	Receives error information.
connectionClosed()	Notifies when TWS terminates the connection.

Ewrapper Functions	Description
updateAccountValue()	Receives current account values.
updateAccountTime()	Receives the last time account information was updated.
updatePortfolio()	Receives current portfolio information.
nextValidId()	Receives the next valid order ID upon connection.
contractDetails()	Receives contract information.
contractDetailsEnd()	Identifies the end of a given contract details request.
bondContractDetails()	Receives bond contract information.
exectDetails()	Receives execution report information.
updateMktDepth()	Receives market depth information.
updateMktDepthL2()	Receives Level II market depth information.
updateNewsBulletin()	Receives IB news bulletins.
managedAccounts()	Receives a list of Financial Advisor (FA) managed accounts.
receiveFA()	Receives FA configuration information.
historicalData()	Receives historical data results.
scannerParameters()	Receives an XML document that describes the valid parameters of a scanner subscription.
scannerData()	Receives market scanner results.
realTimeBar()	Receives real-time bars.
currentTime()	Receives the current system time on the server.
fundamentalData()	Receives Reuters global fundamental market data.

d Instantiate the EClientSocket class. This object will be used to send messages to TWS.

e Call the following functions:

EClientSocket Functions	Description
eConnect()	Connects to TWS.
eDisconnect()	Disconnects from TWS.
reqMktData()	Requests market data.
cancelMktData()	Cancels market data.
reqMktDepth()	Requests market depth.
cancelMktDepth()	Cancels market depth.

EClientSocket Functions	Description
reqContractDetails()	Requests contract details.
placeOrder()	Places an order.
cancelOrder()	Cancels an order.
reqAccountUpdates()	Requests account values, portfolio, and account update time information.
reqExecutions()	Requests a list of the day's execution reports.
reqOpenOrders()	Requests a list of current open orders for the requesting client and associates TWS open orders with the client. The association only occurs if the requesting client has a Client ID of 0.
reqAllOpenOrders()	Requests a list of all open orders.
reqAutoOpenOrders()	Automatically associates a new TWS with the client. The association only occurs if the requesting client has a Client ID of 0.
reqNewsBulletin()	Requests IB news bulletins.
cancelNewsBulletins()	Cancels IB news bulletins.
setServerLogLevel()	Sets the level of API request and processing logging.
reqManagedAccts()	Requests a list of Financial Advisor (FA) managed account codes.
requestFA()	Requests FA configuration information from TWS.
replaceFA()	Modifies FA configuration information from the API.
reqScannerParameters()	Requests an XML document that describes the valid parameters of a scanner subscription.
reqScannerSubscription()	Requests market scanner results.
cancelScannerSubscription()	Cancels a scanner subscription.
reqHistoricalData()	Requests historical data.
cancelHistoricalData()	Cancels historical data.
reqRealTimeBars()	Requests real-time bars.
cancelRealTimeBars()	Cancels real-time bars.
exerciseOptions()	Exercises options.
reqCurrentTime()	Requests the current server time.
serverVersion()	Returns the version of the TWS instance to which the API application is connected.

EClientSocket Functions	Description
TwsConnectionTime()	Returns the time the API application made a connection to TWS.
reqFundamentalData()	Requests Reuters global fundamental data. There must be a subscription to Reuters Fundamental set up in Account Management before you can receive this data.
cancelFundamentalData()	Cancels Reuters global fundamental data.

To run the program, ensure that the TwsSocketClient.dll is in the same directory as your executable, or in your path. By default, the TwsSocketClient.dll file installs into your C:\Windows\system or C:\WINNT\system32 directory.

Chapter 23 - Additional Resources

There are many resources out there that will be adequate in getting you where you need to go. If you have some books or places that you like, feel free to stick with them. The following are the resources we find most helpful, and perhaps they'll be good to you, too!

Help with Microsoft Visual Studio and C++ Programming

While this book is intended for users with C++ programming experience, we understand that even experienced programmers need help every once in a while.

The best place to go to find additional help with C++ programming in Microsoft Visual Studio is in the Help menu in Visual Studio or on Microsoft's web site at http://msdn.microsoft.com/en-us/vstudio/default.aspx. This is the Visual Studio Developer Center, and from here you can access complete information about Visual Studio. You can also find information specific to Microsoft Visual C++ here.

There are literally hundreds of additional printed and web-based resources for C++ programmers. We encourage you to investigate these on your own.

Help with the TWS C++ API

For help specific to the TWS C++ API, the one best place to go, really the ONLY place to go, is the Interactive Brokers website. Once you get there, you have lots of resources. Just type www.interactivebrokers.com in your browser's address line. Now that you're there, let me tell you where you can go.

As of this writing, the IB website looks as I'm describing. IB has a tendency to revamp the look and organization of their site every year or two, so have a little patience if it looks slightly different from what's described here.

The API Reference Guide

The API Reference Guide includes sections for each API technology, including the DDE for Excel. The upper level topics which are shown directly below the main book are applicable across the board to all or multiple platforms.

To access the API Reference Guide from the IB web site, select *API Solutions* from the Trading menu, then click the IB API button, then click the Reference Guide tab. Click the Online API Reference Guide button to open the online guide, which contains a section devoted entirely to the DDE for Excel API.

The API Beta and API Production Release Notes

The beta notes are in a single page file, and include descriptions of any new additions to the API (all platforms) that haven't yet been pushed to production. The API Release Notes opens an index page that includes links to all of the past years' release notes pages. The index provides one-line titles of all the features included in each release.

To access these notes from the IB web site, select *API Solutions* from the Trading menu, then click the IB API button, then click the Release Notes tab and select a link to the latest API

production release notes. You can also access the release notes for the latest API Beta release from this page.

The TWS API Webinars

IB hosts free online webinars through WebEx to help educate their customers and other traders about the IB offerings. They present the API webinar about once per month, and have it recorded on the website for anyone to listen to at any time.

- To register for the API webinar, from the IB web site click Education, then select *Webinars*. Click the Live Webinars button, then click the API tab.
- To view the recorded version of the API webinar, from the Live Webinars page click the Watch Previously Recorded Webinars button. Links to recorded versions of previously recorded webinars are listed on the page.

API Customer Forums

You can trade ideas and send out pleas for help via the IB customer base accessible through both the IB Bulletin Board and the Traders' Chat. The bulletin board includes a thread for the API, and thus provides an ongoing transcript of questions and answers in which you might find the answer to your question. The Traders' Chat is an instant-message type of medium and doesn't retain any record of conversations.

- "To view or participate in the IB Bulletin Board, go to the Education menu and click *Bulletin Boards & Chats*. Click the Bulletin Board tab, then click the Launch IB Discussion Forum button to access all of our bulletin boards, including the TWS API bulletin board.
- To participate in the Traders' Chat, you need to click the Chat icon from the menu bar on TWS. Note that both of these customer forums are for IB customers only.

IB Customer Service

IB customers can also call or email customer service if you can't find the answer to your question. However, IB makes it clear that the APIs are designed for use by programmers and that their support in this area is limited. Still, the customer service crew is very knowledgeable and will do their best to help resolve your issue. Simply send an email to:

api@interactivebrokers.com

IB Features Poll

The IB Features Poll lets IB customers submit suggestions for future product features, and vote and comment on existing suggestions.

From the IB web site, click About IB, then select *New Features Poll*. Suggestions are listed by category; click a plus sign next to a category to view all feature suggestions for that category. To submit a suggestion, click the *Submit Suggestion* link.

Index