patsystems
powers trades. builds exchanges. manages risk.

Client Trading API - Application Developer's Kit

Developer's Guide for API Version 8.8

Version: 2.15.5
Date: 16 October 2014

Any requests for further information or clarification of content should be referred to:

Patsystems (UK) Ltd.
Level 26
30 St Mary Axe
London EC3A 8EP
Tel: +44 (0) 20 7423 6700
Fax: +44 (0) 20 7680 9731

Contents

1 Overview 8

1.1 Introduction 8
1.2 Intended Audience 8
1.3 Patsystems Architecture 9
1.4 Parameter Passing 10
1.5 Function Results 10
1.6 Licensing 11
1.7 Getting further help 11

2 Guidelines for Development 12

2.1 Initial tasks 12
2.2 Environments 13
2.2.1 Test and Live Environments 13
2.2.2 Secure Sockets Layer 14
2.3 Logging in to Patsystems 14
2.4 Reference and Trade Data Downloads 15
2.5 Price & Market Depth Updates 17
2.6 Retrieving Reference and Trade Data 18
2.7 Making Trades 19
2.8 Synthetic Orders 20
2.9 Fills and Positions 20
2.10 Logging off 21
2.11 Scheduled Downtime 22
2.12 Message Alerts 22
2.13 Retrieving Reports 23
2.14 Order Management Integration 23
2.15 Running against the DEMOAPI.DLL 24

3 API Reference 26

3.1 Data Types and Parameters 26
3.2 Setup Functions 26
3.2.1 ptDisable 27
3.2.2 ptDisconnect 27
3.2.3 ptDumpLastError 27
3.2.4 tenable 28
3.2.5 ptForcedLogout (callback) 28
3.2.6 ptGetAPIBuildVersion 29
3.2.7 ptGetConsolidatedPosition 29
3.2.8 ptGetErrorMessage 30
3.2.9 ptHostLinkStateChange (callback) 30
3.2.10 ptInitialise 30
3.2.11 ptLogString 31
3.2.12 ptMemoryWarning (callback) 32
3.2.13 ptNotifyAllMessages 32
3.2.14 ptPriceLinkStateChange (callback) 32
3.2.15 ptPurgeCompleted (Callback) 33
3.2.16 ptReady 33
3.2.17 ptRegisterAtBestCallback 34
3.2.18 ptRegisterBlankPriceCallback 35
3.2.19 ptRegisterCallback 35
3.2.20 ptRegisterCommodityCallback 36
3.2.21 ptRegisterConStatusCallback 37
3.2.22 ptRegisterContractCallback 38

3.2.23 ptRegisterDOMCallback 39
3.2.24 ptRegisterExchangeCallback 40
3.2.25 ptRegisterExchangeRateCallback 40
3.2.26 ptRegisterFillCallback 41
3.2.27 ptRegisterGenericPriceCallback 42
3.2.28 ptRegisterLinkStateCallback 43
3.2.29 ptRegisterMsgCallback 44
3.2.30 ptRegisterAmendFailureCallback 44
3.2.31 ptRegisterOrderCallback 45
3.2.32 ptRegisterOrderQueuedFailureCallback 46
3.2.33 ptRegisterOrderSentFailureCallback 47
3.2.34 ptRegisterOrderCancelFailureCallback 47
3.2.35 ptRegisterOrderTypeUpdateCallback 48
3.2.36 ptRegisterPriceCallback 49
3.2.37 ptRegisterSettlementCallback 50
3.2.38 ptRegisterSubscriberDepthCallback 51
3.2.39 ptRegisterStatusCallback 52
3.2.40 ptRegisterStrategyCreateFailure 53
3.2.41 ptRegisterStrategyCreateSuccess 53
3.2.42 ptRegisterTickerCallback 54
3.2.43 ptRegisterTraderAddedCallback 55
3.2.44 ptSetClientPath 56
3.2.45 ptSetEncryptionCode 56
3.2.46 ptSetHandshakePeriod 57
3.2.47 ptSetHostAddress 57
3.2.48 ptSetHostHandshake 57
3.2.49 ptSetHostReconnect 58
3.2.50 ptSetInternetUser 58
3.2.51 ptSetMemoryWarning 58
3.2.52 ptSetOrderCancelFailureDelay 59
3.2.53 ptSetOrderQueuedFailureDelay 59
3.2.54 ptSetOrderSentFailureDelay 60
3.2.55 ptSetPDDSSL 60
3.2.56 ptSetPDDSSLCertificateName 60
3.2.57 ptSetPDDSSLClientAuthName 61
3.2.58 ptSetPriceAddress 61
3.2.59 ptSetPriceAgeCounter 62
3.2.60 ptSetPriceHandshake 62
3.2.61 ptSetPriceReconnect 63
3.2.62 ptSetSSL 63
3.2.63 ptSetSSLCertificateName 63
3.2.64 ptSetSSLClientAuthName 64
3.2.65 ptSetSuperTAS 64
3.2.66 ptSetMDSToken 65
3.2.67 ptSubscribeBroadcast 65
3.2.68 ptSubscribePrice 65
3.2.69 ptSubscribeToMarket 66
3.2.70 ptSuperTASEnabled 67
3.2.71 ptUnsubscribeBroadcast 67
3.2.72 ptUnsubscribePrice 67
3.2.73 ptUnsubscribeToMarket 68
3.3 Reference Data Functions 70
3.3.1 ptCommodityExists 70
3.3.2 ptCommodityUpdate (Callback) 70
3.3.3 ptContractAdded (callback) 71

3.3.4 ptContractDeleted (callback) 71
3.3.5 ptContractExists 72
3.3.6 ptContractUpdate (Callback) 72
3.3.7 ptCountCommodities 73
3.3.8 ptCountContracts 73
3.3.9 ptCountExchanges 73
3.3.10 ptExchangeUpdate (Callback) 74
3.3.11 ptCountOrderTypes 74
3.3.12 ptCountReportTypes 74
3.3.13 ptCountTraders 75
3.3.14 ptCreateStrategy 75
3.3.15 ptDataDLComplete (callback) 78
3.3.16 ptExchangeExists 78
3.3.17 ptGetCommodity 79
3.3.18 ptGetCommodityByName 80
3.3.19 ptGetContract 81
3.3.20 ptGetContractByExternalID 83
3.3.21 ptGetContractByName 83
3.3.22 ptGetExchange 84
3.3.23 ptGetExchangeByName 85
3.3.24 ptGetExchangeRate 85
3.3.25 ptGetExtendedContract 86
3.3.26 ptGetExtendedContractByName 87
3.3.27 ptGetOptionPremium 88
3.3.28 ptGetOrderType 89
3.3.29 ptGetReport 91
3.3.30 ptGetReportSize 92
3.3.31 ptGetReportType 93
3.3.32 ptGetTrader 94
3.3.33 ptGetTraderByName 95
3.3.34 ptNextOrderSequence 95
3.3.35 ptOrderTypeUpdate (Callback) 95
3.3.36 ptReportTypeExists 96
3.3.37 ptTraderAdded (Callback) 97
3.3.38 ptTraderExists 97
3.4 User Functions 97
3.4.1 ptAcknowledgeUsrMsg 98
3.4.2 ptCountUsrMsgs 98
3.4.3 ptDOMEnabled 98
3.4.4 ptEnabledFunctionality 99
3.4.5 ptGetLogonStatus 100
3.4.6 ptGetUsrMsg 101
3.4.7 ptGetUsrMsgByID 102
3.4.8 ptLockUpdates 103
3.4.9 ptLogOff 103
3.4.10 ptLogon 103
3.4.11 ptLogonStatus (callback) 104
3.4.12 ptMessage (callback) 105
3.4.13 ptOMIEnabled 105
3.4.14 ptPostTradeAmendEnabled 105
3.4.15 ptUnlockUpdates 105
3.5 Trading Functions 106
3.5.1 ptAddAggregateOrder 106
3.5.2 ptAddAAOrder 107
3.5.3 ptAddBasisOrder 108

3.5.4 ptAddBlockOrder 108
3.5.5 ptAddCrossingOrder 108
3.5.6 ptAddCustRequest 109
3.5.7 ptAddOrder 110
3.5.8 ptAddAlgoOrder 114
3.5.9 ptAddOrderEx 114
3.5.10 ptAddProtectionOrder 115
3.5.11 ptAmendOrder 116
3.5.12 ptAmendAlgoOrder 119
3.5.13 ptAtBest (callback) 119
3.5.14 ptBlankPrices 119
3.5.15 ptCancelAll 120
3.5.16 ptCancelBuys 120
3.5.17 ptCancelOrder 121
3.5.18 ptActivateOrder 122
3.5.19 ptDeactivateOrder 122
3.5.20 ptCancelSells 123
3.5.21 ptCountFills 124
3.5.22 ptCountOrderHistory 124
3.5.23 ptCountOrders 125
3.5.24 ptCountContractAtBest 125
3.5.25 ptCountContractSubscriberDepth 126
3.5.26 ptDoneForDay 126
3.5.27 ptFill (callback) 127
3.5.28 ptGetAveragePrice 127
3.5.29 ptGetContractAtBest 128
3.5.30 ptGetContractAtBestPrices 129
3.5.31 ptGetContractPosition 130
3.5.32 ptGetContractSubscriberDepth 131
3.5.33 ptGetFill 132
3.5.34 ptGetFillByID 134
3.5.35 ptGetGenericPrice 134
3.5.36 ptGetOpenPosition 135
3.5.37 ptGetOrder 136
3.6 Order Sub-States 141
3.7 Fill Sub-Types 141
3.8 Global Trading States 141
3.8.1 ptGetOrderEx 142
3.8.2 ptGetOrderByID 142
3.8.3 ptGetOrderByIDEx 142
3.8.4 ptGetOrderHistory 143
3.8.5 ptGetOrderHistoryEx 143
3.8.6 ptGetPrice 144
3.9 PriceDetailStruct 144
3.10 PriceStruct 145
3.10.1 ptGetPriceForContract 147
3.10.2 ptGetTotalPosition 148
3.10.3 ptOrder (callback) 148
3.10.4 ptOrderChecked 149
3.10.5 ptPriceSnapshot 149
3.10.6 ptPriceStep 150
3.10.7 ptPriceUpdate (callback) 151
3.10.8 ptPurge 151
3.10.9 ptQueryOrderStatus 152
3.10.10 ptReParent 152

3.10.11 ptSetUserIDFilter 153
3.10.12 ptSnapdragonEnabled 153
3.10.13 ptStatusChange (callback) 153
3.10.14 ptSubscriberDepthUpdate (callback) 154
3.10.15 ptTicker (callback) 154
3.11 Buying Power Functions 156
3.11.1 ptBuyingPowerRemaining 157
3.11.2 ptBuyingPowerUsed 157
3.11.3 ptMarginForTrade 158
3.11.4 ptOpenPositionExposure 159
3.11.5 ptPLBurnRate 160
3.11.6 ptGetMarginPerLot 161
3.11.7 ptTotalMarginPaid 161

Appendix A – API change history 162

Changes introduced in v1.1 162
Changes introduced in v2.8 162
Changes introduced in v2.8.1 164
Changes introduced in v2.8.2 164
Changes introduced in v2.8.3 164
Changes introduced in v2.8.3-3 164
Changes introduced in v2.8.3-4 164
Changes introduced in v2.8.3-5 165
Changes introduced in v2.8.3-5.2.x 165
Changes introduced in v2.8.3-5.4 165
Changes introduced in v2.8.3-5.6 165
Changes introduced in v6 165
Changes introduced in v6.1 166
Changes introduced in v6.2 167
Changes introduced in v6.3 168
Functional Description for 6.3 API :Connect 9.0 169
Enhanced Automated Price Injection Model (APIM) 169
Inter Commodity Spreads – Strategy Creator 169
Inter Commodity Spreads – Order Entry 169
Inter Commodity Spreads – Order Amend 170
Yesterday's Daily Settlement Price (YDSP) 170
Enhanced Depth Of Market 170
Functional Description for 6.3 API : CME €$ 170
Reduced Tick Contracts 170
Pseudo Level-2 Depth 171
Indicative Prices 171
Enhanced Quote Request – RFQ Entry 171
Enhanced Quote Request – Price Updates. 171
PDD Synchronise Message 171
API Changes 171
Price Blanking 172
Changes introduced in v7.1 172
Diagnostic log file changes 173
Active Market Monitoring 173
Changes introduced in v7.2 173
Functional Description for 7.2 API 174
Changes introduced in v7.3 174
Changes introduced in v7.4 174
Functional Description for 7.4 API 174
Changes Introduced in V7.6 175

Changes Introduced in V7.8 175
Changes Introduced in V7.9.6 175
Changes Introduced in V8.2.1 175
Changes Introduced in V8.4.4 176
Changes Introduced in V8.4.5.2 177
Updates Introduced in V8.4.5.2 177
Updates Introduced in V8.4.6 177
Updates Introduced in V8.4.7 178
Updates Introduced in V8.7 178

Index of Functions 179

Document Control 182

1 Overview

1.1 Introduction

PATS API enables third party applications to interface to the **Professional Automated Trading Systems** (Patsystems) trading engine. The API is provided as a single Dynamic Linked Library, named *PATSAPI.DLL.*

The PATS API requires the following system configuration:

CPU:	1000MHz or greater
Memory:	512 MB
Operating System:	Windows Vista (32 or 64 bit)
	Windows 7 (32 or 64 bit)
	Windows 8

The API provides a series of functions that enable order manipulation and operation. Further functions provide access to reference data.

The API notifies the calling application when events occur (for example, when an order is filled). This event notification is implemented by supplying the API with a callback routine that the API executes on the application's behalf when the event occurs.

Some of these callbacks return variables filled in by the API. Only ordinal or short strings are passed in this manner. Where the data is more complex, the calling application calls a query function to obtain the data. For example, the *order updated* callback will return the O*rder ID*. Further details of the order must be obtained from the API by calling the `query order` function.

1.2 Intended Audience

This document is intended for third party developers who want to use the Patsystems Client Trading API to produce their own front end or interface to a Patsystems trading environment.

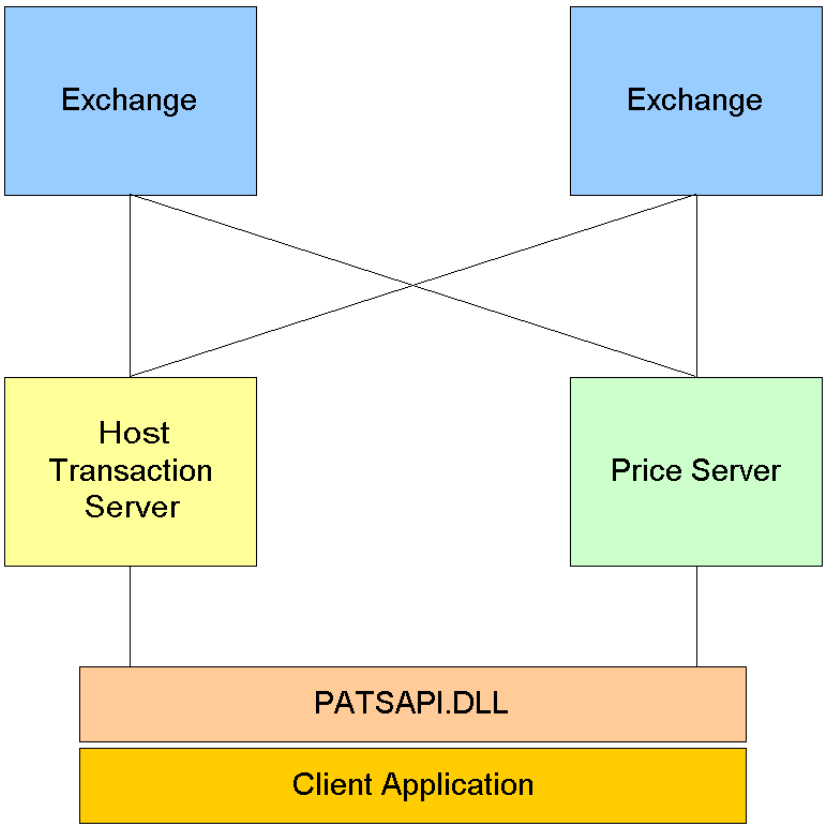
The purpose of this document is to describe the use of the Patsystems Client Trading API for making a trading connection to a Patsystems installation.

This document does not seek to instruct you in designing or writing your own program. Advice is limited to how your program should interact with the Patsystems Client Trading API.

1.3 Patsystems Architecture

A basic understanding of the architecture and terminology used by the Professional Automated Trading System will be beneficial when building the application.

Note that some terms are interchangeable. Although this document seeks to be consistent, you may hear these terms when speaking to Patsystems Support.



The client application uses the PATSAPI.DLL (the *API*) to submit orders, receive fills, and receive prices. The orders are sent to the *Transaction Server* (also called the *Host*) where they undergo validation and are sent to the correct exchange.

Acknowledgements, *Rejections* and *Fills* for these orders are returned by the exchanges to the Transaction Server. They are then routed to the API and notification is given to the client application.

Prices are received from the exchanges and routed to the Price Server (also called the Price Proxy or Price Feed) which directs them to the API only if requested to do so.

1.4 Parameter Passing

The API accepts and returns data in one of thrthe following formats:

1. Binary number
2. Single 1 byte character
3. Null-terminated character string

Normally, multiple fields will be passed or received. These fields are stored in a packed record, that is, fields in the record are not aligned on boundaries, they occupy sequential bytes in memory.

The binary numbers may be single byte or four byte integer in size. The null terminated string is an array starting at element zero and terminated by the null (ASCII 0) character. That is to say, strings are provided in C format.

The parameters passed into the routines are passed either by reference or by immediate value. When a variable is passed by reference, the 32 bit address of the variable appears on the stack. When a variable is passed by immediate value, the actual value appears on the stack. The parameter passing to the API conforms to the following rules.

1. Write access variables are always passed by reference.
2. Integer variables (8 or 32 bit) that are read-only are passed by immediate value.
3. Strings are passed by reference even if they are read-only.
4. Records are passed by reference even if they are read-only.
5.

The Windows API call stack method is used when calling routines. That is, parameters are pushed onto the stack from right to left, registers are not used for parameters and the called routine removes the entries from the stack. Each parameter takes 32 bits regardless of actual size. This means that where an 8 bit integer is passed by immediate value, the entry will occupy the entire 32 bits of the stack entry (although the top 24 bits will be undefined).

1.5 Function Results

Most calls to PATSAPI return a function result as an integer. This value should be examined to decide if the call has succeeded or not. Ignoring the status may lead to unexpected behaviour in your application when communicating with the API.

Some functions do not return results. In most cases, these functions are simple routines requiring little or no validation. Care should still be exercised to ensure that correct data are passed to these routines. Unexpected behaviour may be experienced if invalid information is sent to these routines.

Success is indicated by a result code of zero (equivalent to the Patsystems supplied constant *ptSuccess*). Errors are indicated by a positive result. The actual value returned indicates the error condition. The *ptGetErrorMessage* function can be used to obtain a descriptive error message for the returned value of an API call.

1.6 Licensing

Before you submit your application for conformance testing, choose an *Application ID* for your application and submit it to Patsystems for approval. This *Application ID* should be chosen to clearly represent your company and application. It may be up to 20 characters long.

When your application has passed our conformance tests and is approved for production use, you will be issued with a license key unique to each third party application. The application and license pair identifies the application to the Patsystems trading engine and is used to grant access to specific users of the third party application. It is not possible to use the API to connect to users that have not been authorised with this information.

The license key is not required to connect to the demonstration API and a specific test key will be used to connect to our test environments.
To protect your application from theft, the license details for production connections must be embedded non-visibly in your application. It is not recommended that these details be displayed in free text either on the screen or in a text file.

1.7 Getting further help

You can do much of your initial development using the DEMOAPI.DLL supplied with the kit. However, you may wish to purchase access time to our test servers to obtain a production style response (the *demonstration dll* attempts to simulate live trading scenarios but cannot be expected to match exactly).

To purchase server time please contact devzonesupport@patsystems.com and request access to the servers. During development and while you are supporting your application in production, you may participate in the *Developer Zone* forums at www.patsystems.com and post questions there. Patsystems staff and other developers in the Patsystems API community will read and respond to your questions. To post to these forums, you will need a logon and password.

Finally, if you do not receive a suitably timely response, or need to provide us with diagnostic files, you can contact us by email on devzonesupport@patsystems.com. We may post your mail and respond on the Developer Zone forums if we believe the issue to be relevant to the rest of the developer community.

2 Guidelines for Development

This chapter provides guidelines for developing a trading application using the PATSAPI library. It describes the recommended methods for performing some of the tasks such an application may need. These recommendations are drawn from the experience of writing the **Patsystems** client to use this DLL.

2.1 Initial tasks

There are some initial functions that must be executed before performing any trading using the API. These tasks configure the API for use and perform some basic checks. Follow these steps:

1. The API will generate and use files on a given path. The default path for these files is the path of the executable using the API. To change the path, call *ptSetClientPath* before initializing the API.
2. Initialize API using *ptInitialise*. This initializes the data structures in the API and must be performed before any other steps. It also accepts the application ID and license number used later to verify access to Patsystems.
3. Set any diagnostic information flags using *tenable*. Use these only when initially developing your application, as the functionality can result in large log files (especially price diagnostics).
4. Set SSL and the SSL Certificate Name using *ptSetSSL* and *ptSetSSLCertificateName* if SSL is being used.
5. Set IP configuration details using the following two calls. These calls define the connection details of the Patsystems Transaction Server (Host) and the Patsystems Price Server
    a. *ptSetHostAddress*
    b. *ptSetPriceAddress*
6. Register the required callback routines using *ptRegisterLinkStateCallback* for the host connection and *ptRegisterCallback* to register for *ptLogonStatus* and *ptForcedLogout* callbacks.
7. Register any other optional callback routines
8. Set any other API control parameters. For example, by calling *ptNotifyAllMsgs*
9. Start the API processing by calling *ptReady*

After *ptReady* has been called, the API will attempt to connect to the Host using the IP values previously specified. The IP socket will undergo several rapid state changes before becoming connected. The normal state change sequence is Opened-Connecting-Connected.

If the link fails, it will move from the last state to Closed immediately and then wait for a configurable period before re-attempting the connection. This period has a minimum of 5 seconds.

The callback data structure is defined in the header file.

```
void WINAPI CPatsConnection::OnHostLinkStateChange(LinkStateStructPtr pData)
{
        g_Pats().CheckPatsOrderEntryServerConnectionState(pData);
            }
            .
            .


if( m_Api.ptRegisterLinkStateCallback(ptHostLinkStateChange,
CPatsConnection::OnHostLinkStateChange) != ptSuccess)
        {
                return false;
        }
```

Note: The Price Feed is not connected to at this stage. Prices are not available until after a successful log on.

It is possible to disconnect from the servers and then reconnect without closing the application, using a call to *ptDisconnect*. This will close links to the host and price feed. You can then also change address of either server by making calls to *ptSetHostAddress* or *ptSetPriceAddress*. To re-establish connections, the application must call *ptReady*, followed by a call to *ptLogon*.

2.2 Environments

2.2.1 Test and Live Environments

ptInitialise is for specifying which environment you are connecting to and for controlling certain API behavior. Normal API behavior is specified with either the *ptClient* or *ptTestClient* environments. This will deliver all order state changes, including the Unconfirmed Filled and Unconfirmed Part-Filled states that can result from the Eurex and a/c/e exchanges. The delivery of the Unconfirmed states can be suppressed by specifying the *ptGateway* or *ptTestGateway* environments.

To connect to a test environment, such as out servers in London or Chicago to perform conformance testing, you must use either the *ptTestClient* environment or the *ptTestGateway* environment. Connection to production servers requires using the *ptClient* or *ptGateway* environments. The logon will fail if you try to log on to a test server when you have set a production environment or vice versa.

Connection to the demonstration DLL is available only with the *ptDemoClient* environment.

2.2.2 Secure Sockets Layer

The API will communicate over Secure Sockets Layer as well as regular IP. To do so you will need to know if the remote server is using SSL. If it is, you will need a certificate to install locally – failure to do this will result in the SSL connection being classed as untrusted and the connection will not be made.

Also, you will need to know the SSL Certificate Name – this value is communicated to the Secure Sockets Layer and checked at the server side. If the value is not correct then the connection will not be established.

Errors in creating Secure Sockets Layer connections are logged by the API in PATSDLLerror.log

2.3 Logging in to Patsystems

Before starting to trade, you must complete an application logon to **Patsystems**. In this action the application supplies the Patsystems user name and password for the trader and these are validated on the Patsystems Transaction Server along with the application ID and license number specified in *ptInitialise*. A return status will be returned to your application via the *ptLogonStatus* callback when the logon has been validated.

Logon cannot occur until the API has connected to the Host. This is indicated by the *ptHostLinkStateChange* callback, which will show the old and new states of the IP socket defined as the Host socket in the previous phase.

To complete the logon:

1. Wait for *ptHostLinkStateChange* to show “Connected”.
2. Wait a few seconds (3-5) while API sets encryption detailed structures.
3. Issue logon request by calling *ptLogon*.
4. Wait for *ptLogonStatus* callback to fire.
5. Call *ptGetLogonStatus* to obtain logon status details.
6. If status is *ptLogonSucceeded* then wait for *ptDataDLComplete* callback to fire.

The Patsystems trading engine uses the following information to determine if the log on is allowed:

User Name	
Password	
Application ID	entered in *ptInitialise*
License	entered in *ptInitialise*.
Environment	

If logon was not successful, then the reason will be supplied in the data returned by *ptGetLogonStatus*.

If the logon was successful, the API will attempt to connect to the Price Server using the IP address and socket defined in the set-up phase. The status of this connection will be reported by the *ptPriceLinkStateChange* callback.
Example:

```
if (m_Api.ptRegisterCallback(ptLogonStatus, CPatsConnection::OnLogonStatus) != ptSuccess)
        {
                return false;
        }
if (m_Api.ptReady() != ptSuccess)
        {
                // API not initialised
                return false;
        }

//Set up data structure for logon, read form ini file
LogonStruct logon;
memset(&logon, 0, sizeof(LogonStruct));

strncpy(logon.UserID, m_settings.GetString(strIniFileUserID, "USER").data(),
          sizeof(logon.UserID));
strncpy(logon.Password, m_settings.GetString(strIniFilePasswd, "PASS").data(),
          sizeof(logon.Password));

logon.Reset = 'Y';
//This is waiting for the host socket to connect
DWORD dwWaitTime = WaitForSingleObject(m_hReadyToLogon, 30000);

if( dwWaitTime == WAIT_TIMEOUT )
        {
                return false;
        }
::Sleep(1000); // small delay to assure smooth logon
int nErr = m_Api.ptLogOn(&logon);
if ( nErr != ptSuccess)
        {
                return false;
        }
```

2.4 Reference and Trade Data Downloads

If logon was successful, the API receives its reference data and stores it locally in memory. When this download has completed, the *ptDataDLComplete* callback fires. This event signals that the API is now in a state to process orders and other requests.

However, if reference data has not altered since the last log on, it is not downloaded (the API stores a local copy on exiting). Therefore a varying amount of time may elapse between a successful *ptLogon* and receiving the *ptDataDLComplete* callback. Note that the callback will **always** fire to signal that the reference data is up to date and valid, even if a full download did not occur.

Full reference and trade data is downloaded under the following conditions:

- It is the first logon of the day.
- The username is different from the last username used.
- The “reset” field in *ptLogon* has been set.
- The user’s data has been changed by the system and risk administrator.

The result of this is that if a user logs out of the application and logs back in again using the same user name and the local reference data is believed to be correct then a reload is not received from the transaction server, resulting in a faster reconnection.

Hint! If there has been a connection loss, get the latest guaranteed order states by setting the reset field to `Y`. This is particularly relevant if you have order states that show as *Queued* for a significant time.

Example:

```
void WINAPI CPatsConnection::OnReferenceDataReady()
{
   //The trading data is available, so we can start downloading the
information
   g_Pats().OnTradingDataAvailable();
}

void CPatsConnection::OnTradingDataAvailable()
{
   //The trading data has been downloaded from the Pats server, and is now
   //ready for us to use
   std::cout << "REFDATA: Reference Data has been downloaded" <<
std::endl;
   SetEvent(m_hReferenceDataAvailable);
}
//-----------------------------------------------------
if( m_Api.ptRegisterCallback(ptDataDLComplete,
CPatsConnection::OnReferenceDataReady) != ptSuccess )
   {
           return false;
   }
//We want to wait for the both the logged on event, and the reference data
//available event to occur before we continue with the logon

HANDLE events[2] = { m_hLoggedOn, m_hReferenceDataAvailable};

//Login is synchronous, so block until sequence completes
dwWaitTime = ::WaitForMultipleObjects(2, events, TRUE, 60000); //60 second
timeout

//If we didn't get a response from Pats within 60 seconds, or there
//was an error logging in, then return
if ( dwWaitTime == WAIT_TIMEOUT)
   {
           _ASSERTE(0);
           return false;
   }
```

2.5 Price & Market Depth Updates

The Price Server will not supply any price data until it has been requested by the API. This is not an automatic process and the client application must specifically tell the API what prices it wants to receive. This is done by calling *ptSubscribePrice* which takes the exchange name, contract name and contract date as parameters.

The application can discontinue the supply of prices by calling *ptUnsubscribePrice*. Multiple calls can be made to *ptSubscribePrice*, and the same number of corresponding calls to *ptUnsubscribePrice* are required to cause the price subscription to be discontinued. This allows your application to subscribe and unsubscribe the same price from multiple windows without accidently stopping your price stream while there is still a window requiring prices. For example, if Window A and Window B both subscribe to the Mini S&P, when Window B is closed and the price unsubscribed, the price stream will still supply the Mini S&P prices required by Window A.

Once the Price Server has been notified of which prices to provide, updates to these prices are notified by the *ptPriceUpdate* callback. This will provide the contract that the price applies to and is issued every time a price changes.

To obtain the price details you must call the *ptGetPriceForContract* routine to obtain the price details. This call will return the current price details listed below.

- Bid, Offer
- Implied Bid
- Implied Offer
- Last 20 Trades
- High
- Low
- Opening
- Closing
- Total
- Bid Depth-of-Market 0 through 9
- Offer Depth-of-Market 0 through 9

The volume is returned along with the price if this is appropriate and a price age counter is also provided to show when the price was last updated. The number of seconds before the age counter expires is configurable by calling *ptSetPriceAgeCounter*. If a price update callback executes and this counter is zero, then the age counter has expired. The *ptPriceUpdate* callback is issued when a stale counter expires.
Note that age counters are maintained for all price items, including depth, opening, closing, lows and highs. These all expire and this at first may seem to be unusual but the expiry must be taken in context. For example, a new intra-day high price will shortly expire due to the low frequency of updates.

A direction indicator is also provided with the Price information, indicating the direction of movement from the previous price.

2.6 Retrieving Reference and Trade Data

The PATS API provides access to all reference data required to implement a trading application. This data is stored internally to the API in memory lists, which are indexed from **zero**. This imposes some restrictions on how the data may be accessed while retaining an efficient application.

All general reference data items provide at least two routines:

ptCount*nnnnnn* - returns total number of items in list

ptGet*nnnnnn(x)* - returns a single item from the list by position *x*

This allows reference data to be read from the API by the application using a loop. The *count* function is used to return the total records and this value defines the end of the loop. For each iteration of the loop, the *get* function is used to return an item.
In some cases, filtered access to the list will be provided for the primary key as long as this will return a unique record. In no cases will indexed access be provided with a filter on a non-unique key.

2.7 Making Trades

Once the application has logged on to the host, received the reference data update and subscribed to all the prices it wants, the application is in a position to enter trades into Patsystems. The following two points are key to this process:

- All orders processed by Patsystems are identified by a Patsystems Order ID
- All orders undergo several state changes during their lives.

During its life, the order will undergo a number of state changes, identified by the *State* field returned by *ptGetOrder(ByID)*. These states are defined in the reference section for the *ptGetOrder* routine and the *ptOrder* callback.

Normally, whenever the order undergoes a state change the callback *ptOrder* will fire, returning the Patsystems Order ID of the order that has changed. There are two identifier fields, and old and new order ID. This is used to tie the temporary identifier to the Patsystems order identifier at the point the order goes to the sent state for connections to a standard TAS. Connections to an S-TAS will contain the Patsystems Order ID from the queued order state.

As the order states change, new records are assigned to each order in the list of orders held in the API. The most recent record for each order reflects the most recent state and the earlier ones make up an order history (these history records are held in a separate list for each order). *ptGetOrder(ByID)* will only provide the most recent record pertaining to the order. To obtain historical order information, use the *ptCountOrderHistory* and *ptGetOrderHistory* functions.

Rapid order state changes will trigger the callback for each state change, but you may find that by the time you call the query function to find the new state, the underlying data has been updated to reflect the new order state, leading to the appearance that an order state has been missed. It is important to remember that the query function *ptGetOrder(ByID)* will return the most recent state and the “missing” state will be found in the order history.

As an example, an order might go through the following states in its life:

Queued, Sent, Working, Part Filled, Filled

Existing orders that are still active may be amended using the *ptAmendOrder* call or cancelled using the *ptCancelOrder* routine. The Patsystems order ID is specified to these routines to identify the order. As well as cancelling a specific order, groups of orders may be cancelled using *ptCancelBuys*, *ptCancelSells* and *ptCancelAll*.

2.8 Synthetic Orders

Synthetic orders provide Stop or Stop Limit behavior where an exchange interface does not support Stop or Stop Limit orders. There are two kinds of synthetic orders, but this section is concerned with the ones managed locally within the API. These orders are stored locally in the API until they are triggered by the appropriate price, at which point they are submitted to the transaction server for processing. An Order ID beginning with the letter ‘S’ identifies a synthetic order.

The synthetic ‘S’ Order ID remains while the order is in a held state. The order may be retrieved, cancelled or amended by accessing it using this Order ID. During this time, the Display ID remains blank. When the Order is sent and acknowledged by the transaction server the Order ID is set to the Patsystems Order ID. At this point, the Display ID is set to the Patsystems Order ID and any history records for the order are also updated. An Order callback will be triggered indicating the previous Order ID, and the new Order ID.

Stop Limit orders require a second price. The first price, the trigger price, is placed in the *Price* field. The second price, the limit price, is placed in the *Price2* field. This second price field is not used for market, stop or limit orders.

Synthetic orders are deleted on log out because they are held internally to the API and the act of logging out suggests a lengthy disconnection period will be started.

They are not deleted when calling *ptDisconnect* or when the price feed temporarily disconnects due to a network problem, but be aware that these actions disconnect the price feed that would trigger the order. There is some risk that when the price feed is re-established that the synthetic order will trigger and that this will be later than desired. You may wish to add functionality that detects a lengthy disconnection of the price feed and suggests cancelling synthetic orders.
The alternative to these locally managed orders, which are lost on logout and cannot be shared between users, is to use the Synthetic Order Management System at the clearer. This optional SyOMS server will manage the orders within the server architecture, which means they can be shared and are persisted over a logout. The range of synthetic orders supported by SyOMS is also greater.

2.9 Fills and Positions

Fills may be received in three ways: response to an order, in response to a fill entered by the administrator (external fills) or from **Patsystems** to

show the previous overnight position (netted fills). Fills are notified by the *ptFill* callback which provides the **Patsystems** order ID and the **Patsystems** Fill ID.

If the fill is for an order, the callback contains the OrderID. If the fill is not for an order, then the callback contains the string EXTERNAL or NETTED as appropriate. An EXTERNAL fill is one entered by the Risk Administrator to reflect a trade not done on the **Patsystems** servers. A NETTED fill is the method by which an overnight position is reflected the next day – a fill for the held position will appear with a price of the settlement price of the contract.

Fills are notified by the *ptFill* callback. The order record itself contains the amount filled so far and the average price of the fills. The API records the fill details for each fill as it is received and this detail may be obtained by calling *ptGetFill*. This provides indexed access and is used in conjunction with *ptCountFills* to read the list of all fills. To return fills for an order or contract, the entire list of fills must be read and unwanted records discarded.

A new function *ptGetFillByID* can be used to retrieve the fill that caused the callback to trigger, providing a quick means of obtaining the details as they arrive. Fills are stored in a list sorted by Fill I.D. so it is not possible to assume that new fills appear at index entry “n+1”.

The fill and order states are delivered by different messages and trigger separate callbacks. There will be a message for the order state change to reflect the fill that will trigger the *ptOrder* callback and a message for the fill details to reflect the price and volume of the fill, which will trigger the *ptFill* callback. Patsystems does not guarantee that the fill and the order state change are delivered in any particular order. You might receive the fill callback a fraction before the order state callback, so you must code your application to deal with this potential situation. Also be aware that external fills are delivered by this same mechanism and do not cause an order state change at all.

The API also maintains trading position within contracts. This information can be returned by calling *ptGetOpenPosition* which will return the open profit and the buy/sell position for a trader account within a contract, and *ptGetAveragePrice*, which will return the average prices for the fills making an open position in a contract. A third routine, *ptGetContractPosition*, returns the total profit and total buys and sells for a trader in a specific contract. Closed profit can be calculated by subtracting the open profit (from ptGetOpenPosition) from the total profit (from ptGetContractPosition). Finally, *ptGetTotalPosition* will return the total profit and total buys and sells for the trader over all contracts.

2.10 Logging off

There are two choices for terminating connection to our servers. Your choice will depend on your intentions after this disconnect.
Calling the *ptLogOff* function **disconnects** the user application from the system and saves reference (e.g. contracts, orders) data to disk. This will break the link to the transaction server, and will free the data structures

used in the API and **requires the application to terminate** or otherwise unload the dll. Further calls to the API will return *ptErrNotInitialised* and may have unpredictable behavior. This is a formal means of shutting down your applications completely and is the mechanism we use for our screen based trading front-end, after which we terminate our program. An alternative to calling the logoff routine is to call *ptDisconnect*, which breaks the connection to the server without freeing API structures. After making this call, you can re-enter the IP and socket information by calling the *ptSetHostAddress* and *ptSetPriceAddress* functions, call the *ptReady* function to restart API processing and then log back on again using *ptLogon*. This is a formal means of disconnecting from our servers while leaving your application running and is the mechanism we use for our FIX trading gateway.

Logging off deletes any synthetic orders from the API, but does not issue callbacks to indicate this fact.

2.11 Scheduled Downtime

The Patsystems servers run an End Of Day process each day. The time this process runs varies between connectivity providers, so contact your provider to find out what time your system will be down. For example, many servers run EOD at 4pm Chicago time in the USA, but in the UK this may be done at 10 or 11pm London time.

The EOD process cannot be run while users are connected to the system and any users that are connected will be forced off. This forced logoff behaves the same way as your application calling *ptLogoff* and requires an application termination.

If you wish your application to remain running and reconnect automatically, you must ensure your application calls *ptDisconnect* before EOD reaches the point of sending the forced logoff. EOD is a scheduled activity and will start at a predictable time each day, shortly after the last exchange has been closed for trading.

The system is opened for trading again as a scheduled activity, which also occurs at a predictable time each day. It is okay to attempt a logon before this time as long as it is after the EOD process has started (so you do not receive the forced logout message). If EOD is still in progress then the logon will be rejected with a message of “All users are currently locked out of the system”, but the API will remain in a running state and further logons can be attempted until connection is established. A period of at least 10 seconds is recommended between attempted logons to avoid overloading the servers. A logon will be accepted as soon as EOD has finished and this may occur before the system is opened for trading.

2.12 Message Alerts

The *ptMessage* callback fires to indicate that there are messages or alerts of interest to the user. This callback provides a message ID that can be used to query the API to obtain the text of the message by using

ptGetUsrMsg. Once the message has been viewed, it may be acknowledged by *ptAcknowledgeUsrMsg*.

Messages and alerts are issued for such things as order changes, fill arrival and manually issued messages from the system administrator.

2.13 Retrieving Reports

The **Patsystems** trading engine provides trading reports for each day of the week for each user. The following reports types are implemented. The strings shown below are the correct values to pass into the routines to obtain a report.

“Monday Trades”, “Tuesday Trades”, “Wednesday Trades”, “Thursday Trades”, “Friday Trades”

These Report Type strings are stored internally to the API and can be obtained by calling *ptGetReportType*. The report types are returned in **alphabetical** order, not in day-of-week. That is, query the API for all report types will return “Friday Trades”, “Monday Trades”, “Thursday Trades”, “Tuesday Trades”, “Wednesday Trades”.

These reports are obtained by issuing two calls. First, *ptGetReportSize* is called to get the total size in bytes of the report including the null terminator character. Secondly, *ptGetReport* is called to obtain the data, providing the API with a suitable sized data buffer in which to write the report data.

Obtain a report using the following method:

1. Call *ptGetReportSize* to obtain size of buffer needed.
2. Allocate a contiguous section of memory of the correct size.
3. Call *ptGetReport* passing the address the start of the memory block.

The data returned in the buffer contains the entire text report, containing embedded CR-LF at the end of every line.

2.14 Order Management Integration

Order Management Integration allows the grouping and managing of multiple exchange execution orders to satisfy client requests and aggregations, and is used to assist brokers in managing the large client requests that are worked over an extended duration of the day in multiple execution orders.

If Order Management Integration (OMI) is enabled for the session, the user will have data structures to allow for alternative back office processing. OMI will need to be enabled on the core components, and calling *ptOMIEnabled* will allow the client application to determine if the OMI functionality is enabled.

The following definitions explain the relationship between the different orders:

- **Aggregate**: This is the level at which a trade gets allocated. Therefore if you wish to allocate many orders for one client as one Aggregate you must make sure they are parented to one block. This is held as a typical order structure with the Aggregate order type (ID = 25)
- **Customer Request**: This contains details of a whole order such as buy 10,000 contracts at 101.04. This will be what a trader actually works and will have a price and quantity and buy/sell indicator. Many orders can be aggregated together under one Aggregate Order for allocation purposes.
- **Order**: This is the level at which the order is executed at the exchange (i.e. this is a Patsystems order).

An order has a one-to-many relationship with a customer request, which in turn has a one-to-many relationship with an aggregate order. An example would be a client who called once to work a 10,000-lot order. This would result in one aggregate order (to control allocation), one customer request (to define the specific request to work 10,000 lots) and a number of execution orders sent over the course of the day to the exchange.

2.15 Running against the DEMOAPI.DLL

You can develop your system initially without making a link to the Patsystems development environment by running your application against our demonstration API. This file is released as *DEMOAPI.DLL* and must be loaded instead of PATSAPI.DLL. As a security measure, you must inform the API that your application knows it is talking to the demonstration DLL by setting the *Env* variable in *ptInitialise* to *ptDemoClient*. If this is not done, the logon call will fail.
There is a restriction in the functionality of the DEMOAPI.DLL and it should be used only to gain understanding of how to make the function calls. It does not behave the same way as the production system for business flows. Patsystems **strongly recommends** you request server access time to our test servers in order to continue development using the real PATSAPI.DLL in order to experience correct business flow responses.

The demonstration DLL does not require a license key or application ID. The demonstration DLL is included in the developer's kit and is also available from the developer zone on our website www.patsystems.com. The website also contains a sample client application to run against the DLL.

The demo requires the following data files to simulate the environment, which are included in the kit:

testacct.txt	trader accounts
testcont.csv	commodities
testdate.csv	contract dates
testotype.txt	order types

testreps.txt	reports
testrtype.txt	report types

The DEMO DLL provides full functionality with the following exceptions:

- Logins are never rejected
- Prices are fed at a potentially slower rate than a busy live market
- Orders over 100 lots are rejected
- No other order rejections occur
- Trades are not accepted unless you have subscribed to a price

While it may be possible with the production API to trade without a price feed, this type of application will be disallowed when connected to a cash margining system in this version of the API.
The DEMO DLL provides a simulated price feed and simple trade matching technology:

- The price feed generates prices automatically between the upper and lower limits set in the testdate.csv file.
- Depth Of Market is also randomly generated.
- Trades will be matched if possible, using this price data
- Trades will be reflected in the Depth Of Market

3 API Reference

3.1 Data Types and Parameters

This chapter details the function calls provided by the API. The following conventions appear in the document:

Case sensitivity:	The routine names appear in the DLL export table exactly as they appear in this document. However, case-sensitivity is language dependant and some languages may resolve these references regardless of case.
Type "char"	Single-byte ASCII character.
Type "byte"	Single-byte integer.
Type "integer"	Four-byte integer.
Type "string[n]"	Strings are zero-based arrays of ASCII 1 byte characters, terminated by the null character. Where an array limit "[n]" is specified the array is deemed to be defined as [0..n] of char. Where no array limit is specified, the string may be any length up to 255 as long as it is null terminated.
Type "struct"	Structures are always packed (byte aligned) records containing the preceding data types.
Floating Point Numbers	Floating point numbers are always passed as ASCII strings. Prices include implied decimal places for contracts ticked in fractions (for example, 100.08 is $100^{8/32}$ if the contract is in $^{1/32nds}$).
Immediate mechanism	Applies only to read-only parameters of char or integer type. Immediate passing expects a value on the stack and takes up 32 bits regardless of size. For example, a char or byte will occupy the bottom 8 bits and the remaining 24 bits are ignored.
Reference mechanism	Applies to any write-access parameter and all string or structure parameters. Reference passing expects the address of the variable or structure on the stack.

3.2 Setup Functions

The following functions initialize and define the working parameters of the API, or confirm architecture level settings such as whether the API is connected to a SuperTAS or not. Many of these functions must be completed before calling functions in later sections.

3.2.1 ptDisable

Arguments: Code integer read-only, immediate value
Returns: none
This routine disables the diagnostic option specified in the integer bitmask.

Code This is an integer bitmask where each bit corresponds to a particular debugging option. The options are:

Refer to tenable for details.

3.2.2 ptDisconnect

Arguments: none
Returns: status code integer
This routine disconnects the current Host and Price Feed connections. It is then possible to call ptSetHostAddress and/or ptSetPriceAddress before calling ptReady again to reconnect to the servers. Once reconnected, ptLogon can be called to log back in to the servers.
This function does **not** delete any synthetic orders in the held-order state. However, note that the price feed that triggers these orders has been disconnected and this may lead to unexpected behavior. Reconnection and logon with either a different user or with the same user and the reset flag enabled will delete the synthetic orders by implication – these actions clear the existing order list and reload it from the data sent from the server. As these orders do not exist on the server, they no longer exist. Patsystems recommends that you cancel any synthetic orders before calling ptDisconnect to avoid undesired (delayed) triggering of synthetic orders when the price feed is reconnected, especially if you intend to be disconnected for an extended period.

The function returns the following error codes:

ptSuccess	Successfully disconnected connections.
ptErrNotInitialised	API is not initialised.

3.2.3 ptDumpLastError

Arguments: **none**

Returns: status code integer
This routine causes the API to write debug information to a file (PATSDLLError.log) for the last error that occurred. It is possible for most API routines to return a result of *ptErrUnexpected*. If this occurs, the application should call *ptDumpLastError*
The function returns the following error codes:

ptSuccess	Routine has completed error dump.
ptErrFalse	API is not initialised.

3.2.4 tenable

Arguments: Code integer read-only, immediate value

Returns: none

This routine enables the diagnostic option specified in the integer bitmask. Price diagnostics (bit2) and IP socket flow diagnostics (bit7) should only be enabled to debug your application, since they can adversely affect performance and produce potentially large files if left running for any length of time in a production environment.

Code This is an integer bitmask where each bit corresponds to a particular debugging option. The options are:

bit 0	show program flow and messages
bit 1	show traffic to/from host
bit 2	show traffic to/from price server
bit 3	show depth-of-market flow
bit 4	show order processing
bit 5	write procedure call log on normal exit
bit 6	show calls to API
bit 7	show detailed log of IP socket and locking

Common useful values are decimal 19 (i.e. bits 0,1 and 4) and 83 (i.e. bits 0,1,4 and 6). The value 255 is not recommended for production use, as it will turn on price feed diagnostics. In a live environment this will result in very large log files if prices are subscribed to.

3.2.5 ptForcedLogout (callback)

Arguments: None

The ptForcedLogout callback notifies that the Transaction Server has forced the API to disconnect, and it should not try to reconnect. The application should either close down immediately, or give the user a message before closing down.
The callback must be registered by the *ptRegisterCallback* routine using the Callback ID of *ptForcedLogout*.

This message will be received if the application is left connected while the End Of Day process is being run on the servers.

Receipt of this message requires the API to be unloaded – it is not possible to leave the API up and use the *ptDisconnect* mechanism. If the API is to be left running over EOD, *ptDisconnect* must be called before this callback can be received. EOD runs at a fixed time each day, so this should be possible.

3.2.6 ptGetAPIBuildVersion

Arguments: APIVersion struct writeable, by reference
Returns: Status integer
The *ptGetAPIBuildVersion* routine is used to obtain the build version number of the API. This information may be useful in an “About” box for your application. It is important to know the build version when discussion potential programming errors within our API.

APIVersion A structure of type APIBuildVer, which consists of one string[26] element in which the version information will be supplied as a text string.

This routine returns the following codes:

ptSuccess The data has been returned

3.2.7 ptGetConsolidatedPosition

Arguments: ExchangeName string[11] read-only, by reference

ContractName	string[11]	read-only, by reference
ContractDate	string[51]	read-only, by reference
TraderAccount	string[21]	read-only, by reference
PositionType	integer	read-only, immediate value
Fill	struct	writeable, by reference

Returns: Status Integer
When a contract date expires it is allowed to be purged from memory along with its orders and fills when ptPurge is called. The API therefore consolidates the trader’s fills which allows the API to calculate the trader’s position.

There are two types of consolidated positions: Start of Day and End of Day. The Position type takes in one of two integer values:

- **ptGTStartOfDay = 0** Start of day position for the given contract
- **ptGTEndOfDay = 1** End of day position for the given contract
-

The function returns the following error codes:

ptSuccess	Successful method call
ptErrNotInitialised	API is not initialised
ptErrNotLoggedOn	User is not logged in
ptErrNoData	The contract date specified cannot be found

3.2.8 ptGetErrorMessage

Arguments: ErrorNo: Integer read only, immediate value

Returns: Error message Pointer

The *ptGetErrorMessage* routine is used to obtain a text explanation of a Status code returned by other API routines.

ErrorNo A status code returned by another API routine.

This routine does not return an error code. Make sure that that valid data is passed to the routine. The return value of this routine is the address of a null terminated character string containing a description of the error.

3.2.9 ptHostLinkStateChange (callback)

Arguments: Data struct writeable, by reference

The ptHostLinkStateChange callback notifies that the IP socket has undergone a state change. The old and new states are returned in the data parameter. This routine is provided by the application to be executed by the API whenever the IP link to the Host alters state.

Data Address of a structure of type LinkStateStruct. The application routine will receive the link status details in this parameter. LinkStateStruct is defined as:

OldState	A byte variable containing the last state.
NewState	A byte variable containing the current state.

The link states can be one of:

1. ptLinkOpened - socket created
2. ptLinkConnecting - socket connecting to remote socket
3. ptLinkConnected - socket connected to remote socket
4. ptLinkClosed - socket connection has been closed
5. ptLinkInvalid - unknown or unexpected state

This callback routine must be registered with the *ptRegisterLinkStateCallback* routine.

3.2.10 ptInitialise

Arguments:	Env	char	read-only, immediate value
	APIversion	string	read-only, by reference
	ApplicID	string	read-only, by reference
	ApplicVersion	string	read-only, by reference
	License	string	read-only, by reference
	InitReset	Boolean	read-only, by reference
Returns	status code	integer	

The ptInitialise routine allocates internal data-structures for the API. It also loads the local copy of the reference data (e.g. list of contracts, exchanges, orders, fills), but the reference data is not valid until a logon has occurred. Until this routine is executed, no other calls have any meaning.

Env A single character describing the environment the API is expected to work under. May be one of *ptClient, ptTestClient, ptDemoClient, ptGateway* or *ptTestGateway*.

APIversion Address of a string variable containing the API's version number. This is provided as a check that the application is linked to the expected version of the API.

ApplicID Address of a string variable containing the application ID provided by Patsystems. This information is checked during the *ptLogon* call as the Patsystems trading engine enables the API on a per user basis.

ApplicVersion Address of a string variable containing the version number of the application. This is defined by the external application and is used for reference only.

License Address of a string variable contains the license string provided by Patsystems. This information is checked during the *ptLogon* call because the Patsystems trading engine enables the API on a per user basis. This license is not required to run against the DEMO DLL or our test systems. The license key is issued once your application has passed the conformance test.

InitReset Allows the client to advise the Trading API not to load the contract and order information during the initializing process performed by the API. Setting this to true will reduce the initializing time considerably, but will cause the download time to increase as a result of having to refresh all of the order and fill data.

To protect your application from theft, the license details for production connections must be embedded non visibly in your application. It is unacceptable to display these license details in free text either on the screen or in a text file.

The function returns the following error codes:

ptSuccess API has been initialized.

ptErrNotInitialised API failed to create data structures. Do not use.

ptErrWrongVersion API is not for expected version.

3.2.11 ptLogString

Arguments: DebugStr string[251], read-only by reference

The ptLogString logs the text contained in the DebugStr parameter to the PATSDLLtrace.log file.

The routine returns the following error codes:

ptSuccess The routine made the change successfully.

ptErrNotInitialised API has not been initialized.

3.2.12 ptMemoryWarning (callback)

Arguments: None
The *ptMemoryWarning* callback will trigger when the available memory on the system gets low. The percentage figure that causes this callback to trigger is set by the *ptSetMemoryWarning* call.
The callback must be registered with the *ptRegisterCallback* routine, passing in ID *ptMemoryWarning*.

3.2.13 ptNotifyAllMessages

Arguments: Enabled char immediate value

Returns: status
The *ptNotifyAllMessages* tells the API to issue a callback for any incoming user message, instead of just alert level messages. The default is to issue a callback only if the user message is an alert.

Enabled A char variable containing either Y or N for enable or disable.

The routine returns the following error codes:

ptSuccess The routine made the change successfully.

ptErrNotInitialised API has not been initialized.

3.2.14 ptPriceLinkStateChange (callback)

Arguments: Data struct writeable, by reference
The ptPriceLinkStateChange callback identifies that the IP socket has undergone a state change. The old and new states are returned in the data parameter. This routine is provided by the application to be executed by the API whenever the IP link to the Price Feed alters state.
The routine fires when the API has completed a successful logon via *ptLogon*. No attempt to connect to the Price Server will be made until a successful log on has been achieved.

Data Address of a structure of type *LinkStateStruct*. The application routine will receive the link status details in this parameter. LinkStateStruct is defined as:

OldState	A byte variable containing the last state.
NewState	A byte variable containing the current state.

The link states can be one of:

1. ptLinkOpened - socket created
2. ptLinkConnecting - socket connecting to remote socket
3. ptLinkConnected - socket connected to remote socket
4. ptLinkClosed - socket connection has been closed

5. ptLinkInvalid - unknown or unexpected state

This callback routine must be registered with the *ptRegisterLinkStateCallback* routine.

3.2.15 ptPurgeCompleted (Callback)

Arguments: ExchangeData struct writeable, by reference
The ptPurgeCompleted callback fires when all the expired items under a particular exchange have been purged from memory. ptPurge must be called before purging is initiated.

ExchangeData Address of a structure of type ExchangeUpdStruct containing details about the exchange which has had all its expired contract dates, orders and fills purged from memory:

ExchangeName	A string[11] variable containing the exchange name.

The routine must be registered with the *ptRegisterExchangeCallback* routine.

3.2.16 ptReady

Arguments: none

Returns: status code
Indicates that the application has finished setting up the API parameters. This will trigger the API to connect to the Host which in turn will cause the callback *ptHostLinkStateChange* to fire as the link becomes connected.

A success code from this function does not indicate that the API has connected to the Host. To determine whether the API has connected, examine the data returned by the *ptHostLinkStateChange* callback.

Note that the link to the Price Server is not made at this stage. The connection will not be attempted until *ptLogon* has successfully logged on to the host.

The function returns the following error codes:

ptSuccess	The API has commenced processing.
ptErrCallbackNotSet	One of the required callbacks has not been provided to the API
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.17 ptRegisterAtBestCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterAtBestCallback routine registers a contract callback routine to notify the User of At Best price changes. The callback procedure provided by the application must accept **one** parameter – the address of the structure containing the data.

Some exchanges supply At Best price details, showing individual firm volume at the best bid or offer. Most exchanges do **not** support At Best price data (i.e. individual firm volume). The Sydney Futures Exchange is one exchange that does.

The callback provides the exchange name, contract name and contract date for the contract that has had an At Best price change. The application should then call *ptGetContractAtBest* to obtain the new At Best details (firm, volume, bid or offer) and *ptGetContractAtBestPrices* to obtain the actual At Best prices.

CallbackID	Integer to identify the callback routine being provided. Must be ptAtBestUpdate.
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type AtBestUpdStruct, passed by reference.

This structure contains the exchange name, contract name and contract date for the contract that has had a change in At Best Price.

AtBestUpdStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
PtErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid contract callback.

3.2.18 ptRegisterBlankPriceCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterBlankPriceCallback routine registers a callback routine to notify users of a price blanking message received by the exchange. The callback procedure provided by the application must accept **one** parameter – the address of the structure containing the data. The callback provides the exchange name, an optional contract name and optional contract date for the expiries to be blanked. If all the expiries for a given Exchange or Contract are to have their prices blanked, only the Exchange or Exchange and Contract details will be passed.

CallbackID	Integer to identify the callback routine being provided. Must be ptBlankPrice.
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type BlankPriceStruct, passed by reference.

This structure contains the exchange name, contract name and contract date for the contract that has had a change in At Best Price.

BlankPriceStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
PtErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid contract callback.

3.2.19 ptRegisterCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterCallback routine registers a general callback routine, one that does not return data. In these cases, the application may need to make a further call to the API to obtain the data for the callback.
This routine is used to register the following callbacks:

ptDataDLComplete	Reference data download from Host has completed.
ptLogonStatus	Host has returned a logon status in response to a *ptLogon* call.

ptOrderBookReset	The OrderBook has been reset, and the client needs to clear down the orders and fills prior to receipt of a number of additional items.
ptMemoryWarning	System memory used has risen above the limit.

CallbackID	Integer to identify the callback routine being provided. Use one of *ptDataDLComplete*, *ptLogonStatus* ,*ptForcedLogout* or *ptMemoryWarning*
CBackProc	Address of a procedure that the API will execute. The procedure must accept **no** parameters.

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid general callback.

3.2.20 ptRegisterCommodityCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterCommodityCallback routine registers a callback routine to notify users of a new commodity received by the API after the logon is complete or if an existing commodity has been updated. The callback procedure provided by the application must accept **one** parameter – the address of the structure containing the data. The callback provides the exchange name and contract name for the commodity.

CommodityUpdStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
PtErrNotInitialised	The API has not been initialised by *ptInitialise*.

ptGetCommodityByName needs to be called to get all of the details about the commodity being added or updated.

3.2.21 ptRegisterConStatusCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterConStatusCallback routine registers the callback routine for notifying of a change in connectivity status. The callback procedure provided by the application must accept **one** parameter – the address of the structure containing the data. The application does not pass this structure to the API as the API provides it.

The callback provides details of any changes in connectivity status of the various components of the Patsystems servers. The application is supplied with the new status as part of the callback.

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptConnectivityStatus.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type ConnectivityStatusUpdStruct, passed by reference.

ConnectivityStatusUpdStruct	**read-only, by reference**
DeviceLabel	string[37]
DeviceType	string[4]
Status	string[4]
Severity	string[4]
DeviceName	string[21]
Commentary	string[256]
ExchangeID	string[21]
Owner	string[21]
TimeStamp	string[15]
SystemID	*string[11]*

Field	Description
`DeviceLabel`	Text label representing the specific device.
`DeviceType`	1 – Exchange 2 – ORE 3 – TAS 4 – ESA 5 – MD 6 – SARA 7 – Client 8 – BOF 9 – TSF
`Status`	1 – Running

Field	Description
	2 – Closed 3 - Initialising 4 – Offline
Severity	1 – Information 2 – Warning 3 - Attention 4 - Urgent 5 – Fatal
DeviceName	Name of the specific device in question.
Commentary	Free text, “display friendly” version of the status where appropriate.
ExchangeID	Exchange ID of the device.
Owner	Originator of the status message.
TimeStamp	Date and time that the status message was reported.
SystemID	Globally unique ID identifying the system in which the status message has occured.

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.22 ptRegisterContractCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterContractCallback routine registers a contract callback routine to notify addition or deletion of contracts. The callback procedure provided by the application must accept **one** parameter – the address of the structure containing the data.
The callback provides the exchange name, contract name and contract date for the contract that has been added or deleted. If the callback is for the addition of a contract, the application may then call *ptGetContractByName* or *ptGetExtendedContractByName* to obtain the new contract details.
This routine is used to register the following callbacks:

ptContractAdded	A new contract has been added.
ptContractDeleted	An existing contract has been deleted.

CallbackID Integer to identify the callback routine being provided. Use one of *ptContractAdded* or *ptContractDeleted*.

CBackProc Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type ContractUpdStruct, passed by reference.

This structure contains the exchange name, contract name and contract date for the contract that was added or removed.

ContractUpdStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid callback.

3.2.23 ptRegisterDOMCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterDOMCallback routine registers a contract callback routine to notify the receipt of a Depth Of Market (DOM) message from the price sever. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data.
The callback provides the exchange name, contract name and contract date for the contract that has had a DOM message received.

CallbackID Integer to identify the callback routine being provided. Use ptDOMUpdate

CBackProc Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type DOMUpdStruct, passed by reference.

This structure contains the exchange name, contract name and contract date for the contract referred to by the DOM message.

DOMUpdStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid callback.

3.2.24 ptRegisterExchangeCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterExchangeCallback routine registers a callback routine to notify users of an update to an existing exchange. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The callback provides the exchange name for the exchange.

ExchangeUpdStruct	**read-only, by reference**
ExchangeName:	string[11]

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
PtErrNotInitialised	The API has not been initialised by *ptInitialise*.

ptGetExchangeByName must be called to acquire the full details of the exchange.

3.2.25 ptRegisterExchangeRateCallback

Arguments:	Callback ID value	integer	read-only, immediate
	CBackProc value	address	read-only, immediate
Returns:	status		

The ptRegisterExchangeRateCallback routine registers the callback routine for notification of a change in exchange rates. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the exchange rate name of the exchange rate which has changed. The application is supplied with the new status as part of the callback.

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptExchangeRate.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type ExchangeRateUpdStruct, passed by reference.

This structure contains the exchange rate name for the exchange rate that changed.

ExchangeRateUpdStruct	**read-only, by reference**
Currency	string[11]

The routine will return one of the following error codes:

ptSuccess	The callback was successfully registered.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not valid.

3.2.26 ptRegisterFillCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterFillCallback routine registers the callback routine for notification of a fill. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the order ID and the Fill ID for the fill that was received.

CallbackID An integer identify the callback routine being provided to the API. Must be *ptFill.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type FillUpdStruct, passed by reference.

FillUpdStruct	**read-only, by reference**
OrderID:	string[11]
FillID	string[71]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.27 ptRegisterGenericPriceCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterGenericPriceCallback routine registers the callback routine for notification of receipt of a generic price type. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the price type and value for the price received. The price can be retrieved by calling ptGetGenericPrice.

CallbackID An integer that identifies the callback routine being provided to the API. Must be *ptGenericPriceUpdate.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type GenericPriceStruct, passed by reference.

GenericPriceStruct	**read-only, by reference**
ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]
PriceType:	Integer
BuyOrSell:	Char

Price type is defined as:

Pats Settlement Price	7	The original settlement price that tries to cover all bases
Limit Upper	21	The upper limit price for the market
Limit Lower	22	The lower limit price for the market
Tick Distance	23	the number of ticks away from the reference price that can be traded
Yesterdays Settlement Price	24	The settlement price of yesterdays trading session - used to calculate the change in day price
Todays Settlement Price	25	the settlement price for todays session usually sent out toward the end of the trading session
Int Minute Marker	31	minute marker used during sessions in certain markets to indicate what the final minute marker price will be.

Final Minute Marker	32	The final MM price used to indicate the price everyone will trade at for that market
EFP trade volume	33	a volume traded off the market
EFS trade volume	34	a volume traded off the market
Block trade volume	35	a volume traded off the market
EFP cumulative volume	36	The total volume traded off the market
EFS cumulative volume	37	a volume traded off the market
Block cumulative volume	38	a volume traded off the market

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.28 ptRegisterLinkStateCallback

Arguments:			
	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterLinkStateCallback routine registers the callback routine for notification of a link state change. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.

CallbackID An integer identifying the callback routine being provided to the API. Can be one of *ptHostLinkStateChange* or *ptPriceLinkStateChange*.

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type LinkStateStruct, passed by reference.

LinkStateStruct	**read-only, by reference**
OldState:	byte
NewState	byte

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.

ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.29 ptRegisterMsgCallback

Arguments:	Callback ID value	integer	read-only, immediate
	CBackProc value	address	read-only, immediate

Returns: status

ptRegisterCallback registers the callback routine for notification of user messages (alerts). The callback procedure provided by the application must accept **one** parameter: the address of the array containing the data. The application does not pass this structure to the API as it is provided by the API.

CallbackID	Integer to identify the callback routine being provided. Use the mnemonic code *ptMessage*.
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter of type string[11], passed by reference.
	MsgID : string [11] read-only, by reference

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised.

3.2.30 ptRegisterAmendFailureCallback

Arguments:	CallbackID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterAmendFailureCallback registers the callback routine for notifications of an order amend sending failure within a period of time. Note that this does not necessarily mean the order was not received by the exchange – only that the API has not received notification of the order changing state from sent within the time specified by *pOrderCancelFailureDelay*.
The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API.

The callback provides the order ID for the order that failed to cancel. The application must then call *ptGetOrderByID* to obtain the order details.

CallbackID	An integer to identify the callback routine being provided to the API. Must be *ptOrderAmendFailure.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type OrderUpdStruct, passed by reference.

OrderUpdStruct	**read-only, by reference**
OrderID:	string[11]
OldOrderID	string[11]
OrderStatus	byte
OFSeqNumber	integer
OrderTypeId	integer

The routine will return one of the following error codes:

ptSuccess The callback address was successfully registered by the API.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

3.2.31 ptRegisterOrderCallback

Arguments: Callback ID integer read-only, immediate value
CBackProc address read-only, immediate value

Returns: status

The ptRegisterOrderCallback routine registers the callback routine for notification of an order change. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the order ID for the order that changed, and its previous order ID, before it changed. Additionally there is an OFSeqNumber which is the index of the order update for that particular Order ID, base 1. The application must then call *ptGetOrderByID* to obtain the new details.

CallbackID An integer to identify the callback routine being provided to the API. Must be set to *ptOrder.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type OrderUpdStruct, passed by reference.

OrderUpdStruct	**read-only, by reference**
OrderID:	string[11]
OldOrderID	string[11]
OrderStatus	byte
OFSeqNumber	integer
OrderTypeId	integer

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.32 ptRegisterOrderQueuedFailureCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterOrderQueuedFailureCallback registers the callback routine for notification of an order sending failure within a period of time. Note that this does not necessarily mean the order was not received by the exchange – only that the API has not received notification of the order changing state from sent within the time specified by *ptSetOrderQueuedFailureDelay*.

The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API.

The callback provides the order ID for the order that failed to cancel. The application must then call *ptGetOrderByID* to obtain the order details.

CallbackID An integer to identify the callback routine being provided to the API. Must be *ptOrderQueuedFailure.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type OrderUpdStruct, passed by reference.

OrderUpdStruct	read-only, by reference
OrderID:	string[11]
OldOrderID	string[11]
OrderStatus	byte
OFSeqNumber	integer
OrderTypeId	integer

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.33 ptRegisterOrderSentFailureCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterOrderSentFailureCallback registers the callback routine for notification of an order sending failure within a period of time. Note that this does not necessarily mean the order was not received by the exchange – only that the API has not received notification of the order changing state from sent within the time specified by *ptSetOrderSentFailureDelay*.

The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API.

The callback provides the order ID for the order that failed to cancel. The application must then call *ptGetOrderByID* to obtain the order details.

CallbackID An integer to identify the callback routine being provided to the API. Must be *ptOrderSentFailure.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type OrderUpdStruct, passed by reference.

OrderUpdStruct	read-only, by reference
OrderID:	string[11]
OldOrderID	string[11]
OrderStatus	byte
OFSeqNumber	integer
OrderTypeId	integer

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.34 ptRegisterOrderCancelFailureCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterOrderCancelFailureCallbackroutine registers the callback routine for notification of an order cancellation failure within a period of time. Note that this does not necessarily mean the order did not cancel – only that it failed to cancel within the time specified by *ptSetOrderCancelFailureDelay*.

The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the returned data. The application does not pass this structure to the API.

The callback provides the order ID for the order that failed to cancel. The application must then call *ptGetOrderByID* to obtain the order details.

CallbackID An integer to identify the callback routine being provided to the API. Must be *ptOrderCancelFailure.*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type OrderUpdStruct, passed by reference.

OrderUpdStruct	read-only, by reference
OrderID:	string[11]
OldOrderID	string[11]
OrderStatus	byte
OFSeqNumber	integer
OrderTypeId	integer

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid order callback.

3.2.35 ptRegisterOrderTypeUpdateCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterOrderTypeCallback routine registers the callback routine for notification of an ordertype change. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the configuration details of the ordertype that has changed.

CallbackID An integer to identify the callback routine being provided to the API. Must be *ptPriceUpdate*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type PriceUpdStruct, passed by reference.

This structure contains the exchange name, contract name and date for the price that changed.

OrderTypeStruct	**read-only, by reference**
OrderType	String[11]
ExchangeName	String[11]
OrderTypeID	integer
NumPricesReqd	Byte
NumVolumesReqd	Byte
NumDatesReqd	Byte
AutoCreated	Char
TimeTriggered	Char
RealSynthetic	Char
GTCFlag	Char
TicketType	String[3]
PatsOrderType	Char
AmendOTCount	integer
AlgoXML	String[51]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.36 ptRegisterPriceCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value
Returns:	status		

The ptRegisterPriceCallback routine registers the callback routine for notification of a price change. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.

The callback provides the exchange name, contract name and contract date of the price that has changed. The application must then call *ptGetPriceForContract* to obtain the new price details.

CallbackID	An integer to identify the callback routine being provided to the API. Must be *ptPriceUpdate*
CBackProc	The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type PriceUpdStruct, passed by reference.

This structure contains the exchange name, contract name and date for the price that changed.

PriceUpdStruct	**read-only, by reference**
ExchangeName	string[11]
ContractName:	string[11]
ContractDate	string[51]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.37 ptRegisterSettlementCallback

Arguments: Callback ID integer read-only, immediate value
CBackProc address read-only, immediate value

Returns: status

The ptRegisterSettlementCallback registers the callback routine that will fire whenever a settlement price is received by the API
The callback provides the exchange name, contract name and contract date for the contract that has had a settlement price change. There is no need to call ptGetPriceForContract, as the last two parameters describe the Settlement Price type received, and the value.
The routine expects the following parameters:

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptSettlementCallback*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type SubscriberDepthUpdStruct, passed by reference.

This structure contains the exchange name, contract name and date for the price that changed, along with the new Settlement Price type, price received and the time & date the price was received.

SettlementPriceStruct	**read-only, by reference**
ExchangeName	string[11]
ContractName:	string[11]
ContractDate	string[51]
SettlementType	integer
Price	string[21]
Time	String[7]
Date	String[9]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.38 ptRegisterSubscriberDepthCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterSubscriberDepthCallback registers the callback routine that will fire whenever subscriber depth of market data is updated. Subscriber depth is related to At Best prices, and supplies a firm's volume available at a particular price (in this case, prices other than best bid and best offer). Most exchanges do not supply this information in their trading price feed. One exchange that does is the Sydney Futures Exchange.

The callback provides the exchange name, contract name and contract date for the contract that has had a Subscriber Depth price change. The application should then call *ptGetContractSubscriberDepth* to obtain the new Subscriber Depth details (firm, price volume, bid or offer).

The routine expects the following parameters:

CallbackID	An integer identifying the callback routine being provided to the API. Must be *ptSubscriberDepthUpdate*
CBackProc	The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type SubscriberDepthUpdStruct, passed by reference. This structure contains the exchange name, contract name and date for the price that changed, along with the new Status for the contract date.

SubscriberDepthUpdateStruct	**read-only, by reference**
ExchangeName	string[11]

ContractName	string[11]
ContractDate	string[51]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.39 ptRegisterStatusCallback

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterStatusCallback routine registers the callback routine for notifying of a change in market status of contract dates. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.
The callback provides the exchange name, contract name and contract date of the contract date for which the status has changed. The application is supplied with the new status as part of the callback.

CallbackID	An integer identifying the callback routine being provided to the API. Must be *ptStatusUpdate*
CBackProc	The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type StatusUpdStruct, passed by reference. This structure contains the exchange name, contract name and date for the price that changed, along with the new Status for the contract date.

StatusUpdateStruct	**read-only, by reference**
ExchangeName	string[11]
ContractName:	string[11]
ContractDate	string[51]
Status	Integer

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.40 ptRegisterStrategyCreateFailure

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterStrategyCreateFailure routine registers the callback routine for notifying of a failure to create a strategy passed in using ptCreateStrategy. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.

The callback provides the user name, exchange name, contract name and contract date of the contract date that was not created, plus the reason the contract was not created.

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptStrategyCreateFailure*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type StrategyCreateFailureStruct, passed by reference.

StrategyCreateFailureStruct	**read-only, by reference**
UserName	string[11]
ExchangeName	string[11]
ContractName:	string[11]
ContractDate	string[51]
Text	string[61]

The routine will return one of the following error codes:

ptSuccess	The callback address was successfully registered by the API.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.
ptErrUnknownCallback	The *CallbackID* value was not recognised as a valid link state callback.

3.2.41 ptRegisterStrategyCreateSuccess

Arguments:	Callback ID	integer	read-only, immediate value
	CBackProc	address	read-only, immediate value

Returns: status

The ptRegisterStrategyCreateSuccess routine registers the callback routine for notifying of a successful creation of a strategy passed in using ptCreateStrategy. The callback procedure provided by the application must accept **one** parameter: the address of the structure containing the data. The application does not pass this structure to the API as it is provided by the API.

The callback provides the user name, exchange name, contract name and contract date generated by the client when sent by the API, plus the actual name of the contract generated.

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptStrategyCreateSuccess*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type StrategyCreateFailureStruct, passed by reference.

StrategyCreateSuccessStruct	**read-only, by reference**
UserName	string[11]
ExchangeName	string[11]
ContractName:	string[11]
ReqContractDate	string[51]
GenContractDate	string[51]

The routine will return one of the following error codes:

ptSuccess The callback address was successfully registered by the API.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

ptErrUnknownCallback The *CallbackID* value was not recognised as a valid link state callback.

3.2.42 ptRegisterTickerCallback

The price ticker does not provide a fully functioning ticker. However, when connected to a full rate market data distrubutor it improves the reliability and accurate transmission of best bid/offer and last traded information.

Arguments: Callback ID integer read-only, immediate value
CBackProc address read-only, immediate value

Returns: status

The ptRegisterTickerCallback routine registers the callback routine for notification of a change in contract price. The callback fires whenever a new price is received for any contract.

Note: Be aware, however, that this is not enough to get all price updates from the available contract. A price subscription should also be performed, usually by calling the ptSubscribePrice function.

The callback returns the contract that the price applies to, the new price and also a flag indicating what was updated. This call provides only best bid, best offer and last traded prices along with their associated volumes. The call differs from the ptPriceUpdate callback in that the price data is supplied with the callback, rather than prompting your application to call the API to supply a price. This makes it a suitable mechanism for providing ticker information such as time and sales.

CallbackID An integer identifying the callback routine being provided to the API. Must be *ptTickerUpdate*

CBackProc The address of the callback procedure that the API will execute. This procedure must accept **one** parameter, of type TickerUpdStruct, passed by reference.

This structure contains details about what has changed.

`TickerUpdStruct` is defined as follows:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.
BidPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
BidVolume	An integer variable containing the volume.
OfferPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
OfferVolume	An integer variable containing the volume.
LastPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
LastVolume	An integer variable containing the volume.
Bid	A char variable, either Y or N, to indicate if this message conatins an update to the bid or bid volume.
Offer	A char variable, either Y or N, to indicate if this message conatins an update to the offer or offer volume.
Last	A char variable, either Y or N, to indicate if this message conatins an update to the last or last volume.

The routine will return one of the following error codes:

ptSuccess The callback address was successfully registered by the API.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

ptErrUnknownCallback The *CallbackID* value was not recognised as a valid link state callback

3.2.43 ptRegisterTraderAddedCallback

Arguments: Callback ID integer read-only, immediate value
CBackProc address read-only, immediate value

Returns: status

The ptRegisterTraderAddedCallback routine registers a callback routine to notify users of a new trader received by the API after the logon is complete or if an existing trader has been updated. The callback

procedure provided by the application must accept **one** parameter – the address of the structure containing the data.
The callback provides the configuration for the trader:

TraderAcctStruct	**read-only, by reference**
TraderAccount	String[21]
BackOfficeID	String[21]
Tradable	Char
LossLimit	Integer

The routine returns the following error codes:

ptSuccess	The callback address was successfully registered by the API.
PtErrNotInitialised	The API has not been initialised by *ptInitialise*.

3.2.44 ptSetClientPath

Arguments: Path string read-only, by reference

Returns: none

The ptSetClientPath routine sets the path used by the API to read and write files. By default, the path of the executable using the API is used.
The parameter used to specify the client path should be a null terminated string, and should end with a backslash (“\ “) character.

Caution! This routine does not return any error codes. Make sure that valid information is passed to this routine. Unexpected results could occur if an invalid path is specified.

3.2.45 ptSetEncryptionCode

Arguments: Ecode char read-only, immediate value

Returns: none

The ptSetEncryptionCode routine requests the API to encrypt messages sent to and from the Transaction Server.
To enable encryption of messages, use this routine prior to call ptReady. Currently, the parameter used to specify the encryption code has a valid range of **A** to **E**, which is used internally to the API to determine the encryption method. Leaving this value blank will result in data being transmitted un-encrypted. However, it is not transmitted in free text, as the messages undergo a compression algorithm.
The recommended setting for this function is **A**.

Caution! This routine does not return any error codes. Make sure that valid information is passed to this routine. Unexpected results could occur if an invalid code is specified.

3.2.46 ptSetHandshakePeriod

Arguments: Period Integer read-only, immediate value

Returns: status

The ptSetHandshakePeriod is used to specify the time between handshake messages between connections. The time is measured in seconds, and should be between 0 and 60. A value of zero will disable handshaking.
The handshake timeout will be set to twice that of the handshake period. By default, the handshake interval is 10 seconds and the timeout is 20 seconds.
The routine returns the following error codes:

ptSuccess	The routine was successful.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.

3.2.47 ptSetHostAddress

Arguments:	IPaddress	string	read-only, by reference
	IPsocket	string	read-only, by reference
Returns:	status	integer	

Sets the Host IP address and socket (aka port) to the values specified in the null terminated string parameters. The IP Address string is expected to be the standard IP format of *number.number.number.number.* **Do not insert leading zeros into the string.** It is also possible to enter a host name if your machine has access to a domain name server that can resolve it.
The IP Socket should be in the format of nnnn, although this is not validated.
Bad values will cause the connection to fail when the *ptReady* call is made and will be notified by the *ptHostLinkStateChange* callback. A success status from this call does not mean the IP address is either valid or reachable. It indicates only that the API was able to set the values.

IPaddress	Pass the address of a null terminated string containing the ASCII representation of the IP address including periods.
IPsocket	Pass the address of a null terminated string containing the ASCII representation of the IP socket.

The function returns the following error codes:

ptSuccess	The IP address and port have been set to the value specified.
ptErrNotInitialised	The API has not been initialised by *ptInitialise*.

3.2.48 ptSetHostHandshake

Arguments:	Interval	Integer	readonly, immediate value
	TimeOut	Integer	readonly, immediate value
Returns:	status		

The *ptSetHostHandshake* routine defines the time between handsakes and the length of wait before the API will assume that connection to the Transaction server has been lost.

Interval Pass the interval in seconds by immediate value. A maximum of 900 seconds is imposed;

Timeout Pass the length of time to wait before connection is assumed to be lost. A minimum of twice the interval is imposed. A maximum of 1800 seconds is imposed.

The function returns the following error codes:

ptSuccess The new value has been stored.

ptNotInitialised The API has not been initialised by *ptInitialise*.

3.2.49 ptSetHostReconnect

Arguments: Interval integer read-only, immediate value

Returns: status

The ptSetHostReconnect routine defines the time that the API will wait before attempting to reconnect to the Host transaction server. Calling this routine is optional and if not called, a value of 10 seconds will be used.

Interval Pass the interval in seconds by immediate value. A minimum of five seconds is imposed; a value less than this will be treated as five seconds. There is no maximum value, although high values will affect the ability of the API to recover from network problems.

The function returns the following error codes:

ptSuccess The new value has been stored.

ptNotInitialised The API has not been initialised by *ptInitialise*.

3.2.50 ptSetInternetUser

Arguments: Enable char read-only, immediate value

Returns: status

The ptSetInternetUser routine determines if the connection will be made over the internet, or through a local area network. This will ensure that for internet connections the API will remain connected, and will only use its internal handshaking to determine the state of connection to the servers. By default, it is assumed the user is not an internet user.

The routine returns the following error codes:

ptSuccess The routine was successful.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

3.2.51 ptSetMemoryWarning

Arguments: MemAmount int read-only, immediate value

Returns: status
The ptSetMemoryWarning routine enables code that will monitor the amount of available memory on the machine and trigger the *ptMemoryWarning* callback if used memory rises above the percentage specified by this routine.
The code will check the available/used memory once per second but will issue the callback only once per minute. The callback will trigger once per minute until the used memory drops below the amount specified.

MemAmount An integer value containing the percentage of used memory that will cause the trigger. When total physical memory used becomes greater than this amount, the callback will fire.

Caution! This routine does not return any error codes. Make sure that valid information is passed to this routine. Unexpected results could occur if an invalid code is specified.

3.2.52 ptSetOrderCancelFailureDelay

Arguments: Delay integer read-only, by reference
The ptSetOrderCancelFailureDelay routine sets the delay (in seconds) for the API to wait before issuing an order cancel failure callback. The minimum value is zero seconds, which will turn off this functionality, the maximum value is 3600 seconds (that is, one hour).
This value, when used in conjunction with the callback, can be used to alert the user to a potential loss of connection within the Patsystems servers (e.g. loss of link to exchange).
This routine expects the following parameter:

Delay An integer value containing the number of seconds to wait before issuing the callback.

Caution! This routine does not return any error codes. Make sure valid information is passed to this routine. Unexpected results could occur if an invalid code is specified.

3.2.53 ptSetOrderQueuedFailureDelay

Arguments: Delay integer read-only, by reference
The ptSetOrderQueuedFailureDelay routine sets the delay (in seconds) for the API to wait before issuing an order queued failure callback. The minimum value is zero seconds, which will turn off this functionality, the maximum value is 3600 seconds (that is, one hour).
This value, when used in conjunction with the callback, can be used to alert the user to a potential loss of connection within the Patsystems servers (e.g. loss of link to exchange).
This routine expects the following parameter:

Delay An integer value containing the number of seconds to wait before issuing the callback.

Caution! This routine does not return any error codes. Make sure that valid information is passed to this routine. Unexpected results could occur if an invalid code is specified.

3.2.54 ptSetOrderSentFailureDelay

Arguments: Delay integer read-only, by reference

The ptSetOrderSentFailureDelay routine sets the delay (in seconds) for the API to wait before issuing an order sent failure callback. The minimum value is zero seconds, which will turn off this functionality, the maximum value is 3600 seconds (that is, one hour).
This value, when used in conjunction with the callback, can be used to alert the user to a potential loss of connection within the Patsystems servers (e.g. loss of link to exchange).
This routine expects the following parameter:

Delay An integer value containing the number of seconds to wait before issuing the callback.

Caution! This routine does not return any error codes. Make sure that valid information is passed to this routine. Unexpected results could occur if an invalid code is specified.

3.2.55 ptSetPDDSSL

Arguments: Enabled character read-only, by reference

Returns: Code integer

This function is deprecated as PDD server does not support SSL.
The ptSetPDDSSL routine sets whether or not the API will use Secure Socket Layer encryption to connect to the Price Server. To enable SSL encryption, the argument should be passed as 'Y'. The default mode is to communicate over standard (non-SSL) sockets.
SSL communication can only be used when connecting to a Price server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider.
The routine returns the following error codes:

ptSuccess The request was successful. SSL will be used.

ptErrNotInitialised The API has not been initialised with *ptInitialise*.

3.2.56 ptSetPDDSSLCertificateName

Arguments: CertName string[51] read-only, by reference

Returns: Code integer

(This function is deprecated as PDD server does not support SSL.)
The ptSetSSLCertificateName routine sets the name of the SSL certificate to use with the Secure Socket Layer encryption when connecting to the Price Server. The certificate must be registered on the machine as otherwise the certificate may be regarded as untrusted and the connection will not be established.
SSL communication can only be used when connecting to a Price server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider. For this method to affect the socket connection, ptSetPDDSSL must be called to enable Secure Sockets Layer and the file SSLSocketLib.dll must be placed in the same folder as PATSAPI.dll.

The routine returns the following error codes:

ptSuccess	The certificate name has been set
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.57 ptSetPDDSSLClientAuthName

Arguments:	CertName	string[51]	read-only, by reference
Returns:	Code	integer	

(This function is deprecated as PDD server does not support SSL.)
The ptSetPDDSSLClientAuthName routine sets the authentication name of the SSL certificate to use with the Secure Socket Layer encryption when connecting to the Price Server. The certificate must be registered on the machine as otherwise the certificate may be regarded as untrusted and it will not connect.

This method enables a further level of security above ptSetPDDSSLCertificateName. The client is issued a distinct Certificate, and when a socket connection is made, the name of the certificate is passed to the PDD where that value is compared against those in a certificate store held on the Price server. The main difference between ptSetPDDSSLClientAuthName and ptSetPDDSSLCertificateName is the former is validating the certificate held on the client, whereas the latter is validating against the certificate held on the server.
SSL communication can only be used when connecting to a Price server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider. For this method to affect the socket connection, ptSetPDDSSL and ptSetPDDSSLCertificateName need to have been called previously.
The routine returns the following error codes:

ptSuccess	The request has successful, SSL encryption will be attempted.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.58 ptSetPriceAddress

Arguments:	IPaddress	string	read-only, by reference
	IPSocket	string	read-only, by reference
Returns:	status	integer	

Sets the Price Server IP address and socket (aka port) to the values specified in the null terminated string parameters.
The IP Address string is expected to be in the standard IP format of *number.number.number.number.* **Do not insert leading zeros into the string.** The Windows socket library will set an incorrect socket target address. For example, use “192.168.69.8” not “192.168.069.008”. It is also possible to enter a host name if your machine has access to a Domain Name Server that can resolve this address.
The IP Socket is to be in the format of nnnn, although the API performs no validation of it.
Bad values will cause the connection to fail when the *ptReady* call is made and will be notified by the *ptHostLinkStateChange* callback.

A success status from this call does not mean the IP address is either valid or reachable. It indicates only that the API was able to set the values.

IPaddress Pass the address of a null terminated string containing the ASCII representation of the IP address including periods.

IPsocket Pass the address of a null terminated string containing the ASCII representation of the IP socket.

The function returns the following error codes:

ptSuccess The IP address has been set to the value specified.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

3.2.59 ptSetPriceAgeCounter

Arguments: MaxAge integer read-only, immediate value

Returns: status

The ptSetPriceAgeCounter routine sets the countdown timer value for prices. This integer value is the number of seconds before the price counter expires unless there has been an update. When a price counter expires, the standard price callback *ptPriceUpdate* is issued. Examine the AgeCounter value returned by *ptGetPriceForContract*. If it is zero, then the price has not been updated for MaxAge seconds.

Note that all price items maintain the age counter, including intra-day high and lows, and the opening and closing bids. These price counters may expire as they are not updated very frequently.

MaxAge An integer value representing the number of seconds before a price is considered stale. This must be zero or greater. If it is set to zero, this effectively disables notification of stale prices. The maximum value is 255 seconds.

The function returns the following error codes:

ptSuccess The price age counter has been successfully set.

ptErrNotInitialised The API has not been initialised by *ptInitialise*.

3.2.60 ptSetPriceHandshake

Arguments: Interval: Integer readonly, immediate value
TimeOut: Integer readonly, immediate value

Returns: status

The *ptSetHostHandshake* routine defines the time between handshakes and the length of wait before the API will assume that connection to the Price server has been lost.

Interval Pass the interval in seconds by immediate value. A maximum of 900 seconds is imposed.

Timeout Pass the length of time to wait before connection is assumed to be lost. A minimum of twice the interval, and a maximum of 1800 seconds, is imposed.

The function returns the following error codes:

ptSuccess The handshake parameters have been set.

ptNotInitialised The API has not been initialised by *ptInitialise*.

3.2.61 ptSetPriceReconnect

Arguments: Interval integer read-only, immediate value

Returns: status

The ptSetPriceReconnect routine defines the time that the API will wait before attempting to reconnect to the Price Feed server. Calling this routine is optional and if not called, a value of 10 seconds will be used.

Interval Pass the interval in seconds by immediate value. A minimum of five seconds is imposed. No maximum is set but high values will affect the ability of the API to recover from network problems.

The function returns the following error codes:

ptSuccess The new value has been stored.

ptNotInitialised The API has not been initialised by *ptInitialise*.

3.2.62 ptSetSSL

Arguments: Enabled character read-only, by reference

Returns: Code integer

The ptSetSSL routine sets whether or not the API will use Secure Socket Layer encryption to connect to the Host Transaction Server. To enable SSL encryption, the argument should be passed as 'Y'. The default mode is to communicate over standard (non-SSL) sockets.

SSL communication can only be used when connecting to a host transaction server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider.

The routine returns the following error codes:

ptSuccess The request was successful. SSL will be used.

ptErrNotInitialised The API has not been initialised with *ptInitialise*.

3.2.63 ptSetSSLCertificateName

Arguments: CertName string[51] read-only, by reference

Returns: Code integer

The ptSetSSLCertificateName routine sets the name of the SSL certificate to use with the Secure Socket Layer encryption when connecting to the Host Transaction Server. The certificate must be

registered on the machine as otherwise the certificate may be regarded as untrusted and the connection will not be established.
SSL communication can only be used when connecting to a host transaction server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider. For this method to affect the socket connection, ptSetSSL must be called to enable Secure Sockets Layer and the file SSLSocketLib.dll must be placed in the same folder as PATSAPI.dll.
The routine returns the following error codes:

ptSuccess	The certificate name has been set
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.64 ptSetSSLClientAuthName

Arguments: CertName string[51] read-only, by reference

Returns: Code integer
The ptSetSSLClientAuthName routine sets the authentication name of the SSL certificate to use with the Secure Socket Layer encryption when connecting to the Host Transaction Server. The certificate must be registered on the machine as otherwise the certificate may be regarded as untrusted and it will not connect.

This method enables a further level of security above ptSetSSLCertificateName. The client is issued a distinct Certificate, and when a socket connection is made, the name of the certificate is passed to the STAS where that value is compared against those in a certificate store held on the STAS server. The main difference between ptSetSSLClientAuthName and ptSetSSLCertificateName is the former is validating the certificate held on the client, whereas the latter is validating against the certificate held on the server.
SSL communication can only be used when connecting to a host transaction server that has been enabled for SSL, and by connecting to the port enabled for SSL communication. To determine whether SSL is available to you, contact your connectivity provider. For this method to affect the socket connection, ptSetSSL and ptSetSSLCertificateName need to have been called previously.
The routine returns the following error codes:

ptSuccess	The request has successful, SSL encryption will be attempted.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.65 ptSetSuperTAS

Arguments: Enabled character read-only, by reference

Returns: Code integer
Linux or Solaris “SuperTAS” host transaction servers use a different message protocol from the regular host servers and the API needs to be notified of your intention to connect to one of these by calling this routine and passing the parameter **Y**.

The routine returns the following error codes:

ptSuccess	The API has been set for SuperTAS communications.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.66 ptSetMDSToken

Arguments: MDSToken ANSI String read-only, by reference

Returns: Status

The PDD can have a token enabled that the API has to pass up for a connection to be established. If the PDD has the token enabled, and the API does not pass the correct token, the socket is closed, and no prices are received.

Contact your connectivity provider to establish whether the Market Data Server is using token authentication or not.

ptSuccess	The API has had the token correctly set.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.

3.2.67 ptSubscribeBroadcast

Arguments: ExchangeName string[11] read-only, by reference

Returns: status

The ptSubscribeBroadcast requests the Price Server to supply Subscriber Depth prices – that is, price and volume for a specific firm. Arrival of this information is notified by the *ptSubscriberDepthUpdate* callback.

Subscriber Price information (applying to the SFE) is enabled or disabled on a per exchange basis. Therefore this subscription request does not include contract or maturity information.

ExchangeName	Address of a string variable containing the ASCII name of the exchange.

The routine returns the following error codes:

ptSuccess	The request has been sent to the Price Server.
ptErrUnknownContract	The Exchange, Contract and Date was not recognised.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.
ptErrNotLoggedOn	The API is not currently logged on to the host.

3.2.68 ptSubscribePrice

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference

Returns: status

The ptSubscribePrice routine requests the Price Server to supply a price feed for the instrument passed to it. Updated prices are notified by the *ptPriceUpdate* callback.
It is legitimate to subscribe to prices before actually connecting to the Price Server. On connection or reconnection to the price feed, the API will automatically subscribe to any prices previously subscribed to during this session. Price subscriptions will not be preserved after the API is closed down; it is therefore necessary to subscribe to prices each time the API is invoked.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

The routine returns the following error codes:

ptSuccess	The request has been sent to the Price Server.
ptErrUnknownContract	The Exchange, Contract and Date was not recognised.
ptErrNotInitialised	The API has not been initialised with *ptInitialise*.
ptErrNotLoggedOn	The API is not currently logged on to the host.

3.2.69 ptSubscribeToMarket

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
Returns:	Status	Integer	

The ptSubscribeToMarket routine takes in Exchange, Contract, and Contract Date information and subscribes to RFQ and last price information for the contract or contracts passed. The ExchangeName is the only required field, ContractName is only required if ContractDate is specified, and ContractDate is optional.

ExchangeName Address of a string[11] variable containing the exchange name for the contract date to be queried.

ContractName Address of a string[11] variable that contains the contract name to query, required if a ContractDate is specified.

ContractDate Address of a string[51] variable that contains the contract date of the contract to query.

The routine returns the following error codes:

ptSuccess	Subscription message was successfully sent to the PDD
ptErrNotInitialised	API has not been initialised.

ptErrNotLoggedOn	API is not currently logged on to the host.
PtErrMDSUnavailable	API is not currently connected to a price feed
PtErrUnknownExchange	ExchangeName passed in is not known
PtErrUnknownCommodity	ContractName passed in is not known
PtErrUnknownContract	ContractDate passed in is not known

.

3.2.70 ptSuperTASEnabled

Returns: Code integer

The ptSuperTASEnabled routine returns whether or not the API is enabled to connect to a Super TAS.

The routine returns the following error codes:

ptSuccess	API is currently connected to a SuperTAS
ptErrNotInitialised	API has not been initialised with *ptInitialise*.
ptErrNotLoggedOn	API is not currently logged on to the host.
ptErrNotEnabled	API is not currently enabled to connect to a SuperTAS.

3.2.71 ptUnsubscribeBroadcast

Arguments: ExchangeName string[11] read-only, by reference
Returns: status

The ptUnsubscribePrice requests the Price Server to stop supplying Subscriber Depth prices. Calling this routine will not stop the feed or regular price and depth of market. The effect of this call is limited to the Subscriber Depth (volume for an individual firm).

ExchangeName Sddress of a string variable containing the ASCII name of the exchange.

The routine returns the following error codes:

*ptSuccess*Request has been sent to the Price Server.

ptErrUnknownContract Exchange, Contract and Date not recognised.

*ptErrNotInitialised*API has not been initialised with *ptInitialise*.

ptErrNotLoggedOn API is not currently logged on to the host.

3.2.72 ptUnsubscribePrice

Arguments: ExchangeName string[11] read-only, by reference
ContractName string[11] read-only, by reference
ContractDate string[51] read-only, by reference

Returns: status

The ptUnsubscribePrice routine requests the Price Server to stop supplying a price feed for the instrument passed to it. An internal reference count on each contract date keeps track of how many calls to *ptSubscribePrice* have been made, and how many calls to

ptUnsubscribePrice have been made. When the number of unsubscribes matches the number of subscribes, the unsubscribe will actually occur. This allows you to easily manage multiple windows with the same price on it. For example, Window A and Window B both subscribe to the Mini S&P. When Window B is closed and the price unsubscribed, the price feed will still be delivering the prices needed by Window A.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

The routine returns the following error codes:

ptSuccess	Unsubscribe request has been accepted
ptErrUnknownContract	Exchange, Contract and Date not recognised.
ptErrNotInitialised	API has not been initialised with *ptInitialise*.
ptErrNotLoggedOn	API is not currently logged on to the host.

3.2.73 ptUnsubscribeToMarket

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
Returns:	Status	Integer	

The ptUnsubscribeToMarket routine takes in Exchange, Contract, and Contract Date information and unsubscribes from RFQ and last price information for the contract or contracts passed. The ExchangeName is the only required field, ContractName is only required if ContractDate is specified, and ContractDate is optional.

ExchangeName Address of a string[11] variable containing the exchange name for the contract date to be queried.

ContractName Address of a string[11] variable that contains the contract name to query, required if a ContractDate is specified.

ContractDate Address of a string[51] variable that contains the contract date of the contract to query.

The routine returns the following error codes:

ptSuccess	Unsubscription message was successfully sent to the PDD

ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not currently logged on to the host.
PtErrMDSUnavailable	API is not currently connected to a price feed
PtErrUnknownExchange	ExchangeName passed in is not known
PtErrUnknownCommodity	ContractName passed in is not known
PtErrUnknownContract	ContractDate passed in is not known

3.3 Reference Data Functions

The following functions read the reference data held by the API. Reference data is stored internally in the API, in a number of lists that each start at element zero. Therefore, each type of reference data has at least two routines. One will return the total number of items in the list and the second will provide indexed access to return the *n'th* item from the list.
The API will sometimes provide a simple filtered access to records that can be uniquely identified. However, for efficiency reasons this is only provided where a single record will be returned. There are no routines that provide filtered, indexed access where the filter is not unique, as this would require the API to scan the list each time a record is required. In this case it is more efficient for the application to scan the entire list and discard records it does not want.
Reference data is not valid until the API has logged on to the host and the data download is complete. Before making any calls to obtain reference data, the following operations must have been completed:

1. Attempt log on by calling *ptLogon*.
2. Received log on result notification via *ptLogonStatus* callback.
3. Checked log on status by calling *ptGetLogonStatus*
4. Received *ptDataDLComplete* callback.

3.3.1 ptCommodityExists

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference

Returns: status

The ptCommodityExists routine indicates whether the specified Commodity Name is known to the API. Both fields are required. The content of the fields is case sensitive.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

The routine returns the following error codes:

ptSuccess	Named commodity exists.
ptErrFalse	Named commodity does not exist.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not currently logged on to the host.
ptErrNoData	No Commodity data in the API.

3.3.2 ptCommodityUpdate (Callback)

Arguments: CommodityData struct writeable, by reference

The ptCommodityUpdate callback fires whenever a new commodity is added post logon or an exist commodity's configuration or status is altered. The callback returns the information which uniquely identifies the commodity that has been updated:

CommodityData Address of a structure of type CommodityUpdStruct containing the details about which commodity has been added or updated. CommodityUpdStruct is defined as

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.

The routine must be registered with the *ptRegisterCommodityCallback* routine.

3.3.3 ptContractAdded (callback)

Arguments: ContractData struct writeable, by reference
The ptContractAdded callback fires whenever a new contract is received post logon. The callback returns information that uniquely identifies the contract that has been added.

ContractData Address of a structure of type ContractUpdStruct containing details about what contract was added. ContractUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.

The routine must be registered with the *ptRegisterContractCallback* routine.

3.3.4 ptContractDeleted (callback)

Arguments: ContractData struct writeable, by reference
The ptContractDeleted callback fires whenever a contract is removed post logon. The callback returns information that uniquely identifies the contract that has been removed.

ContractData Address of a structure of type ContractUpdStruct containing details about what contract was removed. ContractUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.

The routine must be registered with the *ptRegisterContractCallback* routine.

3.3.5 ptContractExists

Arguments:	ExchangeName	string[11	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
Returns:	status		

The ptContractExists function indicates whether the API has data for a particular contract. All fields are required and unexpected results may be returned if some fields are not supplied. The content of the fields is case sensitive.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

The routine can return the following error codes:

ptSuccess	Contract is known to the API.
ptErrFalse	Contract is not known to the API.
ptNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not currently have any information about the contract.

3.3.6 ptContractUpdate (Callback)

Arguments: ContractData struct writeable, by reference

The ptContractUpdate callback fires whenever a contract's status or configuration is altered. The callback returns information that uniquely identifies the contract that has been removed.

ContractData Address of a structure of type ContractUpdStruct containing details about which contract was updated. ContractUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.

The routine must be registered with the *ptRegisterContractCallback* routine.

3.3.7 ptCountCommodities

Arguments: count integer writeable, by reference

Returns: status

The ptCountCommodities routine returns the total number of commodities known to the API.

Count Address of an integer variable in which the API will write the result.

The routine returns the following error codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not currently logged on to the host.

3.3.8 ptCountContracts

Arguments: count integer writeable, by reference

Returns: status

The ptCountContracts routine returns the total number of contracts (a.k.a. “contract dates”) known to the API at the time. The returned count may be used to control a loop to read all of the contracts.

Count Address of an integer variable where the API will write the result.

The function returns the following codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the transaction server.

3.3.9 ptCountExchanges

Arguments: Count integer writeable, by reference

Returns: status

The ptCountExchanges routine returns the total number of exchanges known to the PATS. The returned count may be used to control a loop to read all of the exchanges.

Count Pass the address of an integer variable where the API will write the return value.

The function returns the following codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the transaction server.

3.3.10 ptExchangeUpdate (Callback)

Arguments: ExchangeData struct writeable, by reference
The ptExchangeUpdate callback fires whenever a new exchnage is received post logon, or an existing exchange's configuration or status is altered. The callback returns information that uniquely identifies the Exchange that has been added.

ExchangeData Address of a structure of type ExchangeUpdStruct containing details about what exchange was added / updated. ExchangeUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.

The routine must be registered with the *ptRegisterExchangeCallback* routine.

3.3.11 ptCountOrderTypes

Arguments: Count integer writeable, by reference

Returns status
The ptCountOrderTypes routine returns the total number of order types held in the API. The returned count may be used to control a loop to read all of the order types.

Count Address of an integer variable where the API will write the return value.

The function returns the following codes:

ptSuccess	The call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API has not logged on to the transaction server.

3.3.12 ptCountReportTypes

Arguments Count integer writeable, by reference

Returns status
The ptCountReportTypes routine returns the total number of report types held in the API for the user. Currently, this should return 5 as there is one report type for each working day of the week.

Count Address of an integer variable where the API will write the return value.

The function returns the following codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the transaction server.

3.3.13 ptCountTraders

Arguments	Count	integer	writeable, by reference

Returns status

The ptCountTraders routine returns the total number of trading accounts for the user.

Count Address of an integer variable where the API will write the return value.

The function returns the following codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the transaction server.

3.3.14 ptCreateStrategy

Arguments	StrategyCode	char	readonly, immediate
	NoOfLegs	integer	readonly, immediate
	ExchangeName	string[11]	readonly, by reference
	ContractName	string[11]	readonly, by reference
	Legs	struct	readonly, by reference

Returns status

The ptCreateStrategy routine can be used to create strategies if they do not already exist on an exchange. This requires the exchange and the exchange adapter to support creation of strategies, as is the case with Connect. This routine will send a message through the system to the exchange adapter, requesting the strategy to be created.

If the strategy can be created, the *ptContractAdded* callback should fire and the strategy will appear in the list of contracts. If the strategy cannot be created, either because the exchange does not support it or the strategy already exists, the callback will not fire and no new information will appear in the contract list.
There may be a delay between calling this routine and having the strategy created at the exchange. In general, the strategy is created within a few seconds.

StrategyCode A character containing the appropriate strategy code for the strategy, as listed below.

NoOfLegs An integer containing the number of legs in the strategy

ExchangeName Address of a string[11] variable containing the exchange name that the strategy should be created on

ContractName Address of a string[11] variable containing the contract name that the strategy should be created on

Legs Address of a structure, type StrategyLegsStruct, containing the legs making up the strategy. Up to 16 legs can be defined.

StrategyLegsStruct is defined as:

Leg0 – Leg15	Structure of type StratLegStruct

StratLegStruct is defined as:

ContractType	Char variable containing the contract type of the underlying leg (for example “F”: for future, “C” for call or “P” for put)
ContractDate	A string[51] variable containing the Patsystems contract date of the underlying leg.
Price	A string[11] variable containing the price of the underlying option leg, if this is appropriate. Otherwise set it to zero.
Ratio	An integer containing the leg ratio.
ContractName	A string[11] used to describe the Contract for the different legs used for Inter Commodity Strategies.

The following codes may be specified in the StrategyCode parameter for Connect exchanges. The meaning of these strategies is outside the scope of this document.

- ***ptFUT_CALENDAR,***
- ***ptFUT_BUTTERFLY,***
- ***ptFUT_CONDOR,***
- ***ptFUT_STRIP,***
- ***ptFUT_PACK,***
- ***ptFUT_BUNDLE,***
- ***ptFUT_RTS,***
- ***ptOPT_BUTTERFLY,***
- ***ptOPT_SPREAD,***
- ***ptOPT_CALENDAR_SPREAD,***
- ***ptOPT_DIAG_CALENDAR_SPREAD,***
- ***ptOPT_GUTS,***
- ***ptOPT_RATIO_SPREAD,***
- ***ptOPT_IRON_BUTTERFLY,***
- ***ptOPT_COMBO,***
- ***ptOPT_STRANGLE,***
- ***ptOPT_LADDER,***
- ***ptOPT_STRADDLE_CALENDAR_SPREAD,***
- ***ptOPT_DIAG_STRADDLE_CALENDAR_SPREAD,***
- ***ptOPT_STRADDLE,***
- ***ptOPT_CONDOR,***
- ***ptOPT_BOX,***
- ***ptOPT_SYNTHETIC_CONVERSION_REVERSAL,***
- ***ptOPT_CALL_SPREAD_VS_PUT,***
- ***ptOPT_CALL_SPREAD_VS_CALL,***
- ***ptOPT_STRADDLE_VS_OPTION,***
- ***ptVOL_REVERSAL_CONVERSION,***
- ***ptVOL_OPTION,***
- ***ptVOL_LADDER,***
- ***ptVOL_CALL_SPREAD_VS_PUT,***
- ***ptVOL_SPREAD, ptVOL_COMBO,***
- ***ptVOL_PUT_SPREAD_VS_CALL,***
- ***ptVOL_STRADDLE***

The following codes may be specified in the StrategyCode parameter for Eurex MISS exchanges. The meaning of these strategies is outside the scope of this document.

- ***ptDIV_C_CALENDAR,***
- ***ptDIV_C_SPREAD,***
- ***ptDIV_CONVERSION, ptDIV_F_SPREAD,***
- ***ptDIV_P_CALENDAR,***
- ***ptDIV_P_SPREAD, ptDIV_STRADDLE,***
- ***ptDIV_STRANGLE.***

At the time of writing, only Connect exchanges support strategy creation.
This routine returns the following error codes:

ptSuccess	The call succeeded and the request was sent.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptInvalidUnderlying	The underlying legs are not valid.

3.3.15 ptDataDLComplete (callback)

Arguments: none

The ptDataDLComplete callback is executed when the reference data has been downloaded from the Host. This callback should be interpreted to mean that the reference data is now valid and regular processing can now occur.
During the normal log on process, a full data download occurs if any one of the following conditions apply:

- First log on of the day for the user
- Log on for a different user from last time
- Reference data changed on host
- A reset was requested in the *ptLogon* call

If these conditions are not met, a partial download of reference and trade data will be done. This partial download will consist of any data that is new or has been updated since the last logon.
During log on, this callback is executed regardless of whether a full download has actually occurred. If no full download occurred, this signals that the existing reference data loaded from disk is still valid. The process of loading valid reference data during log on is transparent to the calling application, the only visible difference being a slight delay if data is downloaded from the Host.
The application should wait for this callback to be issued before commencing to trade.
The callback must be registered with the *ptRegisterCallback* routine.

3.3.16 ptExchangeExists

Arguments: ExchangeName string[11] read-only, by reference

Returns: status

The ptExchangeExists function indicates whether a particular exchange (market) is known to the API. **This is a case-sensitive test**.

ExchangeName	Address of a string[11] variable containing the exchange name being tested.

The routine can return the following error codes:

ptSuccess	Exchange is known to the API.

ptErrFalse	Exchange not known to the API.
ptNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not currently have any information on exchanges.

3.3.17 ptGetCommodity

Arguments:	Index	integer	read-only, immediate value
	Commodity	Struct	writeable, by reference
Returns:	status		

The ptGetCommodity routine returns a record from the list of commodities known to the API, indexed by the *Index* parameter. The data in the list is stored in alphabetical order.

Index An integer value representing which record to return. Pass in a value between 0 and *ptCountCommodities* – 1 as the data is indexed starting from zero.

Commodity Address of a data structure of type CommodityStruct where the API will write the contract data. This is a packed structure of the following format.

CommodityStruct is defined as:

ExchangeName	A string[11] variable to contain the exchange name. This is one of the values returned by *ptGetExchange*.
ContractName	A string[11] variable to contain the commodity name. This is the same as the *ContractName* returned by *ptGetContract*.
Currency	A string[11] variable to contain the currency the commodity is traded in. This value is used to pass in to *ptGetExchangeRate* to obtain the exchange rate to local currency.
Group	A string[11] variable to contain the commodity group. This value groups similar commodities together and is provided for display purposes only.
OnePoint	A string[11] variable to contain the ASCII representation of the value of one point. This string must be converted into a floating point number.
TicksPerPoint	An integer variable to contain the number of ticks in a point.
TickSize	A string[11] variable to contain the ASCII representation of the tick size. This string must be converted into a floating point number.
GTStatus	The global status of this commodity

Two data items are used to determine a valid price format. They are TicksPerPoint and TickSize and are used as follows:

TicksPerPoint The number of individual increments in a whole point for a price.

TickSize The minimum movement or price step, always a multiple of TicksPerPoint.

The following table provides some examples. In particular, fractional priced contracts on the CBoT.

TicksPerPoint	TickSize	Example
1	1	Whole points: 6500, 6501, 6502
10	0.1	Tenths: 6500.9, 6501.0, 6501.1
10	0.5	Half points: 6500.0, 6500.5, 6501.0
100	0.25	Quarter points: 6500.75, 6501.00, 6501.25
32	0.01	32nds: 102.30, 102.31, 103.00, 103.01
320	0.005	Half 32nds: 102.310, 102.315, 103.000, 103.005
800	0.002	Eigths of a cent: See note below.

Note: Floor traded grains are traded in quarter cents but, since they used to trade in eighths, the prices are denominated in eighths:

The routine can return the following error codes:

ptSuccess The call succeeded and the data was returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData There are no commodities known to the API.

ptErrInvalidIndex Value supplied for the index is out of the range of data.

3.3.18 ptGetCommodityByName

Arguments:

ExchangeName	string[11]	read-only, by reference
ContractName	string[11]	read-only, by reference
Commodity	struct	writeable, by reference

The ptGetCommodityByName routine returns the commodity data for a specified commodity. This routine does not require an index to access the data – it will scan the known commodities until the specified one is matched and then return the data.

ExchangeName Address of a string[11] variable containing the exchange name of the commodity to be queried.

CommodityName Address of a string[11] variable containing the commodity name to be queried.

Commodity Address of a structure of type CommodityStruct to contain the matching commodity details. See ptGetCommodity for details of CommodityStruct.

The routine can return the following error codes:

ptSuccess	Call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	There are no commodities known to the API
ptErrUnknownCommodity	Commodity name given did not match any known records.

3.3.19 ptGetContract

Arguments:	Index	integer	read-only, immediate value
	Contract	struct	writeable, by reference
Returns:	status		

The ptGetContract routine returns contract details from the API, indexed by the *Index* parameter. Data is stored in the API sorted by contract expiry date and then by contract name.

Index	An integer specifying which record to return. Specify a value between 0 and *ptCountContracts* – 1.
Contract	Address of a structure of type ContractStruct where the API will write the contract details.

ContractStruct is defined as:

ContractName	A string[11] variable to contain the name of the contract. This value matches the *CommodityName* value returned by the *ptGetCommodity* routine.
ContractDate	A string[51] variable to contain the contract date. Together with ContractName, this uniquely identifies the contract to PATS.
ExchangeName	A string[11] variable to contain the exchange that the contract is traded on. This matches one of the values returned by *ptGetExchange*.
ExpiryDate	A string[9] variable to contain the contract expiry date in CCYYMMDD format. Data is store sorted primarily by this field.
LastTradeDate	A string[9] variable to contain the date when trading ceases for the contract in CCYYMMDD format.
NumberOfLegs	Integer. The number of Legs in the Contract.
TicksPerPoint	Integer used to describe the Ticks Per Point for the expiry
TickSize	A string[11] variable used to determing the TickSize for the expiry
Tradable	A char used to indicate if the Contract is available to trade by the user, or if the contract can only be used as reference data.
GTStatus	Integer for the Global Status of a contract
Margin	String[21] Margin per lot – used in risk calculations
ESATemplate	Char - Whether the contract is template or not
MarketRef	String[17] – exchange reference for the contract
InExchangeName	String[11] – exchange which this contract links to *
InContractName	String[11] – exchange which this contract links to *
InContractDate	String[51] – exchange which this contract links to *
ExternalID	An array of 2 LegStruct structures (defined below) containing the first exchange specific contract specification.

LegStruct is defined as an array of 5 string[11] variables. The variables are:

1	The contract type. A string[11] variable, containing for example “F” for future, or “EF” for calendar spread.
2	The exchange commodity name. A string[11] variable that identifies the contract on the exchange, for example ZB, NQ or FDAX.
3	The exchange maturity code. A string[11] variable that identifies the maturity on the exchange. For example, 200212. Format varies by exchange.
4	The strike price. A string[11] variable that identifies the option strike price. Blank for futures.
5	A string[11] variable that further identifies the contract on the exchange. Varies by exchange and is often blank.

* - these fields are currently only used in Settlement and Minute markets

Note: In C or C++ the above indices are zero based, so the first Leg is defined by ExternalID[0], and the Contract Type of the first Leg will be defined by ExternalID[0][0].

For multi leg contracts, i.e. those contracts that have more than 2 legs, it will be necessary to call ptGetExtendedContract to obtain all exchange specific contract specifications.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any contract information.
ptErrInvalidIndex	The specified index does not refer to a valid record.

3.3.20 ptGetContractByExternalID

Arguments:	ContractIn	struct	read-only, by reference
	ContractOut	struct	writeable, by reference
Returns:	status		

The ptGetContractByExternalID routine returns contract details from the API for a contract date with the given external ID fields.

ContractIn Address of a structure of type ContractStruct. The values of ExternalID[1][1], ExternalID[1][2], ExternalID[1][3], ExternalID[2][2] and ExternalID[2][3] are used to find a matching contract.

ContractOut Address of a structure of type ContractStruct where the API will write the contract details. See *ptGetContract* for a description of ContractStruct.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrInvalidIndex	The values specified do not refer to a valid contract record.

3.3.21 ptGetContractByName

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	Contract	struct	writeable, by reference
Returns:	status		

The ptGetContractByName routine returns contract details from the API for a given exchange, contract name and date.

ExchangeName A string specifying the name of exchange to which the contract belongs.

ContractName A string specifying the name of the contract.

ContractDate A string specifying the date of the contract.

Contract Address of a structure of type ContractStruct where the API will write the contract details. See *ptGetContract* for a description of ContractStruct.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any contract information.
ptErrInvalidIndex	The values specified do not refer to a valid contract record.

3.3.22 ptGetExchange

Arguments: value	Index	integer	read-only, immediate
	ExchangeDetails	struct	writeable, by reference

Returns: status

The ptGetExchange routine returns an exchange name from the list of exchanges, indexed by the *Index* parameter. Data is stored in the API alphabetically by ExchangeName.

Index The index into the list, passed by immediate value. Enter a value between 0 and *ptCountExchanges* – 1.

ExchangeDetails Address of a structure of type *ExchangeStruct* where the API can write the exchange data.

Name	A string[11] variable containing the exchange name.
QueryEnabled	A char variable, set to Y if the exchange supports order querying. If not enabled, the *ptQueryOrderStatus* function will return an error.
AmendEnabled	A char variable, set to Y if the exchange supports order amendment.
Strategy	Integer Value used to determine if the Strategy Creator tool is available to the J-Trader client application.
CustomDecs	Char Boolean ('Y' = true) used to determine if the exchange has a default number of decimal places, which can be used to override the price detail being received in price messages.
Decimals	If CustomDecs is true, this integer field returns the number of decimal places to be used as a default for prices received from the exchange
TicketType	Char value used by J-Trader to determine if the exchange has any behaviour specific to that exchange (in J-Trader, this includes the appearance of exchange specific trade tickets, such as FX tickets).
RFQA	Char Boolean value used to determine if the exchange supports RFQ Accept orders.

RFQT	Char Boolean indicating if the exchange supports RFQ Tickdown orders.
EnableBlock	Char Boolean value used to determine if the exchange supports Block trades.
EnableBasis	Char Boolean value used to determine if the exchange supports Basis trades.
EnableAA	Char Boolean value used to determine if the exchange supports Against Actuals trades.
EnableCross	Char Boolean value used to determine if the exchange supports Crossing trades.
GTStatus	Integer variable denoting the Global Status of the exchange

The routine will return the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrInvalidIndex	The index value specified does not refer to a valid record.

3.3.23 ptGetExchangeByName

Arguments: ExchangeName string[11] read-only, by reference
ExchangeDetails struct writeable, by reference

Returns: status
The ptGetExchangeByName routine returns exchange data for an exchange.

ExchangeName	Address of a string[11] variable containing the exchange name to query on.
ExchangeDetails	Address of a structure of type *ExchangeStruct* where the API can write the exchange data.

Refer to ptGetExchange for details of *ExchangeStruct*.

The routine will return the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownExchange	The specified exchange is not recognised.

3.3.24 ptGetExchangeRate

Arguments: Currency string[11] read-only, by reference

ExchRate string[21] writeable, by reference

The ptGetExchangeRate routine returns the exchange rate for converting a currency into the system local currency. The local currency is defined by the Host Administrator and is fixed for each transaction server.

Currency Address of a string[11] variable containing the currency name of a particular commodity. This is value returned in the *Currency* field from *ptGetCommodity*.

ExchRate Address of a string[21] variable where the API will write the exchange rate as an ASCII string. This result must be converted to a floating point number.

Note: The exchange rate is the divisor rate, where *Contract P&L* divided by *Rate* equals *System P&L*.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API is not logged on to the host.
ptErrNoData	The API does not currently hold any exchange rate details.
ptErrUnknownCurrency	The currency code specified was not recognised.

3.3.25 ptGetExtendedContract

Arguments:	Index	integer	read-only, immediate value
	ExtContract	struct	writeable, by reference
Returns:	status		

This function returns the same data as *ptGetContract* except that this routine can return data for up to 16 different legs. The *ptGetContract* routine is limited to two legs (i.e. calendar spreads).

The ptGetExtendedContract routine returns contract details from the API, indexed by the *Index* parameter. Data is stored in the API sorted by contract expiry date and then by contract name.

Index An integer specifying which record to return. Specify a value between 0 and *ptCountContracts* – 1.

ExtContract Address of a structure of type ExtendedContractStruct where the API will write the contract details. ExtendedContractStruct is an extension of ContractStruct providing external ID information for a total of 16 legs. See *ptGetContract* for a description of ContractStruct.

ExtendedContractStruct is defined as:

ContractName	A string[11] variable to contain the name of the contract. This value matches the *CommodityName* value returned by the *ptGetCommodity* routine.
ContractDate	A string[51] variable to contain the contract date. Together with ContractName, this uniquely identifies the contract to **Patsystems**
ExchangeName	A string[11] variable to contain the exchange that the contract is traded on. This matches one of the values returned by *ptGetExchange*.
ExpiryDate	A string[9] variable to contain the contract expiry date in CCYYMMDD format. Data is returned sorted primarily by this field.
LastTradeDate	A string[9] variable to contain the date when trading ceases for the contract in CCYYMMDD format.
NumberOfLegs	Integer. The number of Legs in the Contract.
TicksPerPoint	Integer used to describe the Ticks Per Point for the expiry
TickSize	A string[11] variable used to determing the TickSize for the expiry
Tradable	A char used to indicate if the Contract is available to trade by the user, or if the contract can only be used as reference data.
GTStatus	Integer for the Global Status of a contract
Margin	String[21] Margin per lot – used in risk calculations
ESATemplate	Char - Whether the contract is template or not
MarketRef	String[17] – exchange reference for the contract,
InExchangeName	String[11] – exchange which this contract links to *
InContractName	String[11] – exchange which this contract links to *
InContractDate	String[51] – exchange which this contract links to *
ExternalID	An array of 16 LegStruct structures (refer to *ptGetContract*).

* - these fields are currently only used for Settlement and Minute Markets

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any contract information.
ptErrInvalidIndex	The specified index does not refer to a valid record.

3.3.26 ptGetExtendedContractByName

Arguments:	ExchangeName	string	read-only, by reference
	ContractName	string	read-only, by reference
	ContractDate	string	read-only, by reference
	ExtContract	struct	writeable, by reference

Returns: status

This function returns the same data as *ptGetContractByName* except that this routine can return data for up to 16 different legs. The *ptGetContractByName* routine is limited to two legs (i.e. calendar spreads).
The ptGetExtendedContractByName routine returns contract details from the API for a given exchange, contract name and date.

ExchangeName A string[11] specifying the name of exchange to which the contract belongs.

ContractName A string[11] specifying the name of the contract.

ContractDate A string[51] specifying the date of the contract.

ExtContract Address of a structure of type ExtendedContractStruct where the API will write the contract details. ExtendedContractStruct is an extension of ContractStruct providing external ID information for a total of 16 legs. See *ptGetContract* for a description of ContractStruct and *ptGetExtendedContract* for a description of ExtendedContractStruct.

The routine returns the following error codes:

ptSuccess The call succeeded and the data was returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData The API does not currently hold any contract information.

ptErrInvalidIndex The values specified do not refer to a valid contract record.

3.3.27 ptGetOptionPremium

Arguments:	ExchangeName	string	read-only, by reference
	ContractName	string	read-only, by reference
	ContractDate	string	read-only, by reference
	BuySell	char	immediate value
	Price	string	read-only by reference
	Lots	integer	immediate value
	OPr	string	write by reference
Returns:	status		

This function calculates the option premium required to place a proposed order specified by the parameters passed in the arguments. If BuySell argument passed in is a sell, the premium returned is 0. Otherwise the premium is determined by the below equation:

```
Premium = Lots x <value of 1 point> x Price / <any currency conversion>
```

ExchangeName A string[11] specifying the name of exchange to which the contract belongs.

ContractName A string[11] specifying the name of the contract.

ContractDate	A string[51] specifying the date of the contract.
BuySell	A character value B or S to indicate the buy or sell of the proposed trade.
Price	A string[21] value containing the price of the proposed trade.
Lots	An integer variable passed by immediate value, containing the number of lots for the proposed trade.
Opr	A string[21] variable containing the option premium for the proposed trade.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownContract	The contract described by the arguments is invalid

3.3.28 ptGetOrderType

Arguments:	Index	integer	read-only, immediate value
	OrderTypeRec	struct	writeable, by reference
	AmendOrderTypes	Struct	writeable, by reference
Returns:	status		

The ptGetOrderType routine is designed to allow your application to determine the valid order types that the API will accept. Order types are configured when the host is installed and are not alterable afterwards. Order types will differ between exchanges.
The ptGetOrderType routine returns data on a specific Order Type, as specified by the index. By using the index to retrieve all records, you will have a list of all the valid order types for all the exchanges.

Index	An integer index between 0 and *ptCountOrderTypes* – 1.
OrderTypeRec	Address of a data structure of type OrderTypeStruct where the API will write the returned data. OrderTypeStruct is defined as:

OrderType	A string[11] variable containing the order type.
Exchange	A string[11] variable containing the exchange where this order type is traded.
OrderTypeId	An integer variable containing an enumeration of the possible order types.
NumPricesReqd	A byte variable containing the number of prices required for the Order Type.

NumVolumesReqd	A byte variable containing the number of volumes required for the Order Type.
NumDatesReqd	A byte variable containing the number of dates required for the Order Type.
AutoCreated	A char variable indicating that the API created the order type.
TimeTriggered	A char variable indicating whether the order type has a triggered by a time value
RealSynthetic	A char variable indicating whether the order type is an exchange supported synthetic order type
GTCFlag	A char variable indicating whether the order type is a GTC type
TicketType	A string[3] variable containing the ticket type to be used for that order type. Relevant to Patsystems GUIs only.
PatsOrderType	A Char containing the Pats Order Type
AmendOTCount	An Integer containing the number of amendable order types
AlgoXML	String[51] containing the ALGO XML information

AmendOrderTypes Address of a data structure of type AmendTypesArray where the API will write the returned data. AmendTypesArray is defined as:

AmendTypesArray	A string[501] containing the amendable order types splitted by quotes

The following table lists the current enumerations supported by the API. This list is subject to expansion as new order types are added to the list of supported types – you may receive an integer value not found in the enumeration list.
Current enumerations are fixed and are defined in the header file released with the kit. New order types will be added to the end of the list.
Not all order types are supported on all exchanges, so check the Exchange field returned in the data structure. For example, Eurex supports a “Stop” but Liffe does not (on Liffe, you will see the Synthetic Stop order type).
Order types can be disabled on a per user basis, to prevent specific users from trading specific order types such as GTCs. This leads to the possibility that one user in a TAG is permissioned for an order type and another user is not. The user not permissioned still has the right to see orders of this type, but not submit trades using the order type. In this circumstance, the order type will appear in the second user’s list of order types but with the AutoCreated flag set to ‘Y’, indicating that the order type was created by the API in order to view the order.
Even though the order type appears in the list, this user will not be allowed to trade it.

Valid order types are:

OrderTypeId	Meaning
ptOrderTypeMarket	Market

OrderTypeId	Meaning
ptOrderTypeLimit	Limit
ptOrderTypeLimitFAK	Limit Fill and Kill
ptOrderTypeLimitFOK	Limit Fill or Kill
ptOrderTypeStop	Stop (exchange supported)
ptOrderTypeSynthStop	Synthetic Stop
ptOrderTypeSynthStopLimit	Synthetic Stop Limit
ptOrderTypeMIT	Market if Touched (exch supported)
ptOrderTypeSynthMIT	Synthetic Market if Touched
ptOrderTypeMarketFOK	Market Fill or Kill
ptOrderTypeMOO	Market on Open
ptCrossingBatchType	Crossing Order
ptBasisBatchType	Basis Batch Order
ptBlockBatchType	Block wholesale trade order
ptAABatchType	Against Actual trade order
ptOrderTypeIOC	Immediate or Cancel
ptOrderTypeStopRise	Stop Rise
ptOrderTypeStopFall	Stop Fall
ptOrderTypeRFQ	Request for Quote
ptOrderTypeStopLoss	Stop Loss

The routine returns the following error codes:

ptSuccess	The routine succeeded and the data was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any OrderType data.
ptErrInvalidIndex	The index value does not refer to a valid record.

3.3.29 ptGetReport

Arguments:	ReportType	string[21]	read-only, by reference
	BufferSize	integer	read-only, immediate value
	BufferAddr	address	writeable, immediate value
Returns:	status		

This function is only valid if the reports were requested using the *Reports* flag in the *ptLogon* function.
The ptGetReport routine returns the report data for a particular report type. Reports vary greatly in size, therefore there is no fixed data structure to hold the return data. Rather, the calling application must provide the address of a data area that is large enough to hold the resulting report. The application must call *ptGetReportSize* to obtain the size of this buffer before calling this routine.

ReportType Address of a string[21] variable containing the report type to be returned. This is one of the values returned by *ptGetReportType*: “Monday Trades”, “Tuesday Trades”, “Wednesday Trades”, “Thursday Trades” or “Friday Trades”. The parameter value is case-sensitive.

BufferSize An integer containing the total size of the buffer provided by the calling application. This value is used to check that the report will actually fit. If set correctly (using *sizeof* or similar) it should avoid the API overwriting program space in the calling application.

BufferAddr A pointer to the start of the data structure where the API will write the report data. The API will move *ptGetReportSize* bytes of data into memory commencing at this address.

The routine returns the following error codes:

ptSuccess	The call succeeded and the report data was written to memory.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API is not logged on to the host.
ptErrNoData	The API does not currently hold any report data.
ptErrNoReport	The specified report was not found.
ptErrBufferOverflow	There was not sufficient room to write the data. Either the value passed in *BufferSize* was smaller than the report size or there was an error writing data to the area specified by *BufferAddr*.

3.3.30 ptGetReportSize

Arguments:	ReportType	string[21]	read-only, by reference
	ReportSize	integer	writeable, by reference
Returns:	status		

This function is only valid if the reports were requested using the *Reports* flag in the *ptLogon* function.
The ptGetReportSize routine returns the buffer size required to hold a particular report. This routine must be called before calling *ptGetReport* so a buffer of the correct size can be created and passed to it.

ReportType Address of a string[21] variable containing the report type to be retrieved. This is one of the values returned by *ptGetReportType*: “Monday Trades”, “Tuesday Trades”, “Wednesday Trades”, “Thursday Trades” or “Friday Trades”. The parameter value is case-sensitive.

ReportSize Address of an integer variable where the API will write the buffer size required to hold the report, including the null terminator character. If there is no matching report then this value will be zero and the error *ptErrNoReport* will be returned.

The routine returns the following error codes:

ptSuccess	The call succeeded and the report size was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API is not logged on to the host.
ptErrNoData	The API does not currently hold any report data.
ptErrNoReport	The specified report was not found.

3.3.31 ptGetReportType

Arguments:	Index	integer	read-only, immediate value
	ReportType	string[21]	writeable, by reference
Returns:	status		

This function is only valid if the reports were requested using the *Reports* flag in the *ptLogon* function.

The ptGetReportType routine returns the available report types supplied by the API. There is one report for each day of the week: “Monday Trades”, “Tuesday Trades”, “Wednesday Trades”, “Thursday Trades” and “Friday Trades”. This data is returned in **alphabetical** order not in the order of the days of the week.

Index An integer value indicating which record in the list should be returned. Supply a value between 0 and *ptCountReportTypes* - 1.

ReportType Address of a string[21] variable where the API will write the report type as described above.

The routine returns the following error codes:

ptSuccess	Call succeeded and a report type was returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not contain any report type data.
ptErrInvalidIndex	The index value is outside the valid range and does not point to a record.

3.3.32 ptGetTrader

Arguments:	Index	integer	read-only, immediate value
	TraderAccountRec	struct	writeable, by reference
Returns:	status		

The ptGetTrader routine returns trader account details from the API indexed by the *Index* parameter. Data is stored in the API sorted in alphabetical Trader Account order.

Index An integer value specifying which record is to be returned. Supply a value between 0 and *ptCountTraders* – 1.

TraderAccountRec Address of a data structure of type TraderAcctStruct where the API will write the trader details. The structure is defined as:

TraderAccount	A string[21] variable for receiving the trader account name. This value is the display format of the trader account. It is this value that other routines require when expecting a trader account.
BackOfficeID	A string[21] variable for receiving the back office format of the trader account.
Tradeable	A character containing the value 'Y' if the Account is tradable, otherwise the Trader Account is only valid for viewing.
LossLimit	An integer containing the Loss Limit of a trader account.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any trader account details.
ptErrInvalidIndex	The index is out of range.

3.3.33 ptGetTraderByName

Arguments: TraderAccount string[21] read-only, immediate value

TraderAccountRec struct writeable, by reference

Returns: status

The ptGetTraderByName routine returns trader account details from the API for the Trader Account with the supplied Trader Account name.

TraderAccount Address of a string[21] variable containing the Trader Account name of the Trader Account to retrieve.

TraderAccountRec Address of a data structure of type TraderAcctStruct where the API will write the trader details. See *ptGetTrader* for details of TraderAcctStruct.

The routine returns the following error codes:

ptSuccess	The call succeeded and the data was returned.
ptNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API does not currently hold any trader account details.
ptErrUnknownAccount	The supplied TraderAccount name does not refer to a valid record.

3.3.34 ptNextOrderSequence

Arguments: Sequence integer write, by reference
Returns: no return code

The ptNextOrderSequence returns the next available order number. This will be the order number that the API will use when ptAddOrder is next called.

Sequence Address of an integer variable where the next order number will be written.

The routine returns the following error codes:

ptSuccess	The next sequence number was returned by the API
ptErrNotInitialised	API has not been initialised.

3.3.35 ptOrderTypeUpdate (Callback)

Arguments: OrderTypeData struct writeable, by reference

The ptOrderTypeUpdate callback fires whenever a new order type is added post logon or an existing order type's configuration is updated. The callback returns the order type information:

OrderTypeData Address of a structure of type OrderTypeStruct containing the details about the order type.

OrerTypeStruct is defined as:

OrderType	A string[11] variable containing the order type.
ExchangeName	A string[11] variable containing the exchange where this order type is traded.
OrderTypeId	An integer variable containing an enumeration of the possible order types.
NumPricesReqd	A byte variable containing the number of prices required for the Order Type.
NumVolumesReqd	A byte variable containing the number of volumes required for the Order Type.
NumDatesReqd	A byte variable containing the number of dates required for the Order Type.
AutoCreated	A char variable indicating that the API created the order type.
TimeTriggered	A char variable indicating whether the order type has a triggered by a time value
RealSynthetic	A char variable indicating whether the order type is an exchange supported synthetic order type
GTCFlag	A char variable indicating whether the order type is a GTC type
TicketType	A string[3] variable containing the ticket type to be used for that order type. Relevant to Patsystems GUI's only.
PatsOrderType	A Char containing the Pats Order Type
AmendOTCount	An Integer containing the number of amendable order types
AlgoXML	String[51] cotaning the ALGO XML information

This must be registered by calling ptRegisterOrderTypeUpdateCallback

3.3.36 ptReportTypeExists

Arguments: ReportType string[21] read-only, by reference

Returns: status

The ptReportTypeExists routine indicates whether a particular report type is known to the API.

ReportType Address of a string[21] variable containing the report type to be retrieved. This is one of the values returned by *ptGetReportType*: "Monday Trades", "Tuesday Trades", "Wednesday Trades", "Thursday Trades" or "Friday Trades". The parameter value is case-sensitive.

The routine returns the following error codes:

ptSuccess Order type was recognised by the API.

ptErrFalse Order type is not known to the API.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData API has no report type details at this time.

3.3.37 ptTraderAdded (Callback)

Arguments: TraderData struct writeable, by reference

The ptTraderAdded callback fires whenever a new trader account is added post logon or an existing trader's configuration is updated. The callback returns the trader account information that has been added / updated:

TraderData Address of a structure of type TraderAcctStruct containing the details about which trader has been added or updated.

TraderAcctStruct is defined as:

TraderName	A string[21] variable containing the trader name.
BackOfficeID	A string[21] variable containing the unique ID for the trader
Tradable	Boolean Char variable containing whether the account can trade
LossLimit	Integer variable of the limit on the trader account

This must be registered by calling ptRegisterTraderAddedCallback

3.3.38 ptTraderExists

Arguments: TraderAccount string[21] read-only, by reference

Returns: status

The ptTraderExists routine indicates whether a particular trader account is known to the API. A trader account will only be valid if the user has been set up by the System Administrator to have access to the account. It is possible for the account to be defined on the servers but for the user not to have access to it and, in this case, the routine will return false.
A failure of this test may indicate that the account does not exist at all, or that the account exists but the user is not allowed access to it. The API does not distinguish between these situations.

TraderAccount Address of a string[21] variable containing the trader account to be checked.

The routine returns the following error codes:

ptSuccess User is allowed access to the trader account.

ptErrFalse User is not allowed access to the trader account.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData API does not currently have any trader account details.

3.4 User Functions

This section covers the routines used to control the user access to the API, such as logging on and receiving user messages.

3.4.1 ptAcknowledgeUsrMsg

Arguments: MsgID string[11] read-only, by reference

Returns: status

The ptAcknowledgeUsrMsg routine clears a message notified by the *ptMessage* callback. All alert messages should be acknowledged in this manner, but failure to acknowledge a message will not effect the execution of the API. Once a message of any description has been acknowledged, its status is altered from “Pending” to “Cleared”.

MsgID Address of a string[11] variable containing the message ID (a.k.a. the “sequence” number) of the message to be acknowledged. This value is provided by the *ptMessage* callback.

The routine returns the following error codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the host.
ptErrNoData	API holds no user messages at this time.
ptErrUnknownMsgID	Specified message ID does not refer to a valid message.

3.4.2 ptCountUsrMsgs

Arguments	Count	integer	writeable, by reference
Returns	status		

The ptCountUsrMsgs routine returns the total number of messages (alerts and normal messages) for the user. This value increases throughout the day but is reset after the end of each day.

Count Address of an integer variable where the API will write the return value.

The function returns the following codes:

ptSuccess	Call succeeded.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the server.

3.4.3 ptDOMEnabled

Arguments: none

Returns: status

This routine refers to the availability of the Depth of Market (DOM) data, and is superseded by the *ptEnabledFunctionality* routine and should not be used.

ptSuccess	Call succeeded and the data has been returned.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnexpected	An unexpected error occurred.

3.4.4 ptEnabledFunctionality

Arguments: FunctionalityEnabled integer writeable, by reference
SoftwareEnabled integer writeable, by reference

Returns: Status

The *ptEnabledFunctionality* function returns what functionality and third party software are enabled for this user. Bits are numbers from least significant (0) to most significant (7). This routine is designed for use by J-Trader produced by Patsystems and has no meaning to third-party developers.

FunctionalityEnabled Address of an integer variable where the API will write the bitmask listing what functionality has been enabled by the System Administrator.

SoftwareEnabled Address of an integer variable where the API will write the bitmask listing what third party software has been enabled by the System Administrator. The third party application can use the flag to control how different functionalties are enabled for different users.

FunctionalityEnabled		SoftwareEnabled	
Bit	**Meaning**	**Bit**	**Meaning**
0	DOM enabled	0	User defined bit
1	Post trade amend enabled	1	User defined bit
2	MEL enabled	2	User defined bit
3	Not used.	3	User defined bit
4	PIG enabled	4	not in use
5	Options enabled	5	not in use
6	Strategy creation enabled	6	not in use
7	not in use	7	not in use

For example, if FunctionalityEnabled=31, then DOM, post-trade, MEL and PIG are enabled.

2**0 = 1 (DOM)
2**1 = 2 (post-trade)
2**2 = 4 (MEL)
2**4 = 16 (PIG)
31

To use the user defined region [0..3] of Software Enabled, you will need contact Patsystems and have these codes enabled and attached to a specific user role. This feature is available for applications that wish to have a closer and more integrated relationship with Patsystems. Contact api_tech@patsystems.com for more information.

The routine returns the following error codes:

ptSuccess	The routine has successfully returned the functionality and software enabled.
ptErrUnexpected	An unexpected error occurred.

3.4.5 ptGetLogonStatus

Arguments: LogonStatus Struct writeable, by reference

Returns: status

The ptGetLogonStatus routine returns the current logon status. This routine is called to determine whether the user logon was successful and is called in response to the callback event *ptLogonStatus.* The normal logon sequence is to call *ptLogon* to send the logon message, wait for *ptLogonStatus* callback to fire and then call *ptGetLogonStatus* to find the result of the logon.

LogonStatus Address of a data structure of type LogonStatusStruct where the API will write the last known logon details. The LogonStatusStruct is a packed structure defined as follows:

Status	A byte integer indicating the success or failure of the logon.
Reason	A string[61] field containing the reason for the log on failure or the forced log off. For a successful log on it will contain the string “You are now logged on to PATS”
DefaultTraderAccount	A string[21] field containing the user’s default trader account as set up by the System and Risk Administrator. For a failed log on, this field is blank.
ShowReason	Char boolean – ‘Y’ if we set the reason field
DOMEnabled	Char boolean – ‘Y’ if DOM is enabled
PostTradeAmend	Char boolean – ‘Y’ if Post Trade Amend is set
UserName	String[256] field containing the username
GTEnabled	Char boolean – ‘Y’ if logged into a Global Trading host

The LogonStatus field contains one of:

ptLogonFailed	This value is no longer returned in the production dll.
ptLogonSucceeded	You are now logged on to PATS
ptForcedOut	The host forced the application to be logged off
ptObsoleteVers	The API version passed in to *ptInitialise* is no longer supported.
ptWrongEnv	The connection is for production and this is the test environment, or vice versa
ptDatabaseErr	The core server could not attached to the database
ptInvalidUser	Username is not set up on the system
ptLogonRejected	The username is correct but the logon could not be completed (e.g. wrong password, disabled)

ptInvalidAppl	The application or license details are not correct
ptLoggedOn	This username is already logged on elsewhere
ptInvalidLogonState	Unexpected data was returned from the server

Note: The DEMO DLL may still return you *ptLogonFailed* under some circumstances.

The routine returns the following error codes:

ptSuccess Call succeeded and the data has been returned. This does **not** indicate that the log on was successful. Refer to the *Status* field for this information.

ptNotInitialised API has not been initialised.

3.4.6 ptGetUsrMsg

Arguments: Index integer read-only, immediate value

UserMsg struct writeable, by reference

Returns: status

The ptGetUsrMsg routine retrieves a user message (a.k.a. an "alert") from the list of all messages received so far today. This list grows during the day and is reset during the end-of-day processing after trading closes. Data is returned in message ID order, which is the order in which the messages were created during the day.
The message status is an indication of whether the user has acknowledged the message. It is expected that each message will be acknowledged by a call to *ptAcknowledgeUsrMsg*.

Index An integer specifying the record to return. Supply a value between 0 and *ptCountUsrMsgs* – 1. There should always be at least one message in the queue, which is a notification that the end-of-day procedure was completed.

UserMsg Address of a data structure of type MessageStruct where the API will write the user message details.

The MessageStruct is defined as:

MsgID	A string[11] variable to contain the message ID (a.k.a. the "sequence") that uniquely identifies this message.
MsgText	A string[501] variable to contain the message text for display.
IsAlert	A character variable indicating whether the message is an alert or not. One of "Y" or "N".
Status	The current status of this message. One of "P" - pending "C" - cleared

The routine returns the following error codes:

ptSuccess Call succeeded and the data has been returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API has no user messages currently stored.
ptErrInvalidIndex	The index value specified does not refer to a valid record.

3.4.7 ptGetUsrMsgByID

Arguments:	MsgID	string[10]	read-only, by reference
	UserMsg	struct	writeable, by reference
Returns:	status		

The ptGetUsrMsgBy ID routine retrieves a particular user message (a.k.a. an “alert”) from the list of all messages received so far today. This list grows during the day and is reset during the end-of-day processing. The routine reads directly on the MsgID parameter. This value matches the value handed to the application by the *ptMessage* callback.
The message status is an indication of whether the user has acknowledged the message. It is expected that each message will be acknowledged by a call to *ptAcknowledgeUsrMsg*.

MsgID	Address of a string[10] variable containing the message ID (a.k.a. the “sequence” number).
UserMsg	Address of a data structure of type MessageStruct where the API will write the user message details. The MessageStruct is defined as:

MsgID	A string[11] variable to contain the message ID (a.k.a. the “sequence”) that uniquely identifies this message.
MsgText	A string[501] variable to contain the message text for display.
IsAlert	A character variable indicating whether the message is an alert or not. One of “Y” or “N”.
Status	The current status of this message. One of “P” - pending “C” - cleared

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API has no user messages currently stored.
ptErrUnknownMsgID	The message ID did not match any known message.

3.4.8 ptLockUpdates

Arguments: none

Returns: status

ptLockUpdates queues the updates received from the STAS. This is used to prevent the loss of data if updates are received and the client collects the session data while more contracts arrive, therefore the extra information is not collected as the client does not process callbacks until initialisation is complete.
The routine returns the following error codes:

ptSuccess	The call succeeded
ptErrNotInitialised	API has not been initialised.

3.4.9 ptLogOff

Arguments: none

Returns: status

LogOff **disconnects** the user application from the system and saves reference (for example, contracts, orders) data to disk. This will break the link to the transaction server, free data structures used in the API and **require the application to terminate** or otherwise unload the dll. Further calls to the API will return *ptErrNotInitialised*.
An alternative to calling the logoff routine is to call *ptDisconnect*, which breaks the connection to the server without destroying the API data structures.

This routine deletes any synthetic orders that were in the held-order state. The assumption is that the application wishes to make a permanent disconnection from the system.
The routine returns the following error codes:

ptSuccess	The call succeeded, the log off message was sent – unload the API.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.

3.4.10 ptLogon

Arguments: LogonDetails struct read-only, by reference

Returns: status code

The ptLogon routine sends a log on message to the Host. It does not wait for a reply – once the message has been sent, the routine exits. The expected outcome of issuing this call is for the *ptLogonStatus* callback to be triggered, notifying the application that the log on has been processed and the result (logged on or failed) is now available. Be aware after receiving the connected callback you must wait 2-4 seconds before calling ptLogon.
During this phase, the **Patsystems** trading engine compares the username, password, application ID and license details to determine if the user is allowed to log in from this source. The license details were specified in the *ptInitialise* call and are supplied by **Patsystems**.

As well as the *ptLogonStatus* callback executing, if the log on was successful, the *ptDataDLComplete* callback fires. This signals that the reference data is now valid and the API is ready to process orders.
Once the log on is successful and download is completed, the API initiates a connection to the Price Server using the details set up with *ptSetPriceAddress*. The success of this connection will be reported by the callback *ptPriceLinkStateChange*.
The UserID password can be altered by this call. If the *NewPassword* field is non-blank, then the password for the user will be set to this new value if, and only if, the log on is successful. That is, the value of *Password* must be correct before the value of *NewPassword* is used to change the user's password on the host. If a password is accidentally changed, the system and risk administrator can alter any password.

LogonDetails Address of a structure of type LogonStruct containing the username and password for logging on to PATS. LogonStruct is defined as:

UserID	A string[256] variable containing the **Patsystems** user name.
Password	A string[256] variable containing the password for the user ID.
NewPassword	A string[256] variable containing the new password for the user ID. If this is left blank or is set to null, then the password will not be changed. If set, must be an alphanumeric.
Reset	A character variable controlling whether a complete data download is wanted. To force a data download, set this field to 'Y'
Reports	To force a report download, set this field to 'Y'. If set to 'N' then reports are not available through the ptGetReport funcions.
OTP	A string[21] variable containing the One Time Password (OTP).

Use of Reset Field
NOTE: Version 8.4.6 and up the reset option is no longer used.
Use of OTP Field
The One Time Password is required if this feature is enabled in the Patsystems Core Server.
The routine returns the following error codes:

ptSuccess The call succeeded and the message was sent. This does **not** imply that the log on succeeded – call *ptGetLogonStatus* to determine this.

ptErrNotInitialised API has not been initialised.

ptErrBadPassword The value for *NewPassword* is not valid.

3.4.11 ptLogonStatus (callback)

Arguments: none
The ptLogonStatus callback executes when the Host has processed the log on request sent using *ptLogon*.
When this callback executes it is necessary to obtain the log on status using the *ptGetLogonStatus* function. This signals that the log on was processed and a response is available, it does not signify that the log on succeeded.
The callback must be registered using the *ptRegisterCallback* routine.

3.4.12 ptMessage (callback)

Arguments: MsgID string[11] writeable, by reference
The ptMessage callback fires whenever a user alert is received by the API. The user messages viewed in order provide an audit trail of actions throughout the day. Optionally, this callback will fire when any user message is received, not just alert messages. This is controlled by the *ptNotifyAllMsgs* routine.
The callback hands the message identification string (a.k.a. the "sequence" number) to the calling application. This value can be used in the *ptGetUsrMsgByID* routine to obtain the message text.
Each alert is expected to be acknowledged by the application, to set the state to "Cleared". This is done using the *ptAcknowledgeUsrMsg* routine.

MsgID A string[11] variable passed by reference. This will contain the message ID to pass to *ptGetUsrMsgByID*.

The routine must be registered with the *ptRegisterMsgCallback* routine.

3.4.13 ptOMIEnabled

Arguments: Enabled char read-only, immediate value

Returns: status

The ptOMIEnabled indicates whether the currently logged on user is allowed access to the Order Management Integration routines such as *ptAddAggregateOrder*. Refer to section 2.14 Order Management Integration for more information.
The routine returns the following error codes:

ptSuccess	The user is enabled for OMI
ptErrNotInitialised	API has not been initialised.

3.4.14 ptPostTradeAmendEnabled

Arguments: none

Returns: status
This routine is superseded by the *ptEnabledFunctionality* routine.

3.4.15 ptUnlockUpdates

Arguments: none

Returns: status

ptUnlockUpdates unloads the queued list of updates and sends the updates to the client.
The routine returns the following error codes:

ptSuccess	The call succeeded
ptErrNotInitialised	API has not been initialised.

3.5 Trading Functions

The following routines are used to manipulate the API for trading and process the results.

3.5.1 ptAddAggregateOrder

Arguments:	NewAggOrder	struct	read-only, by reference
	OrderID	string[11]	writeable, by reference
Returns:	status code		

The ptAddAggregateOrder routine is designed to assist in managing multiple exchange orders under one care or master order. The aggregate order is first entered into the system along with the total required volume and subsequent exchange orders are added with *ptAddOrder*. If the aggregate order's Order ID is supplied with these exchange orders, then the underlying exchange orders will be managed within the context of this aggregate care order.

NewAggOrder Address of a structure of type NewAggOrderStruct containing the order details.

OrderID Address of a string[11] variable which will receive the Order ID of the new Order.

If Order Management Integeration (OMI) is enabled for the session, the user will have data structures to allow for alternative back office processing. OMI will need to be enabled on the core components, and calling ptOMIEnabled will allow the client application to determine if the OMI functionality is enabled.

Aggregate:
This is the level at which a trade gets allocated. Therefore if you wish to allocate many orders for one client as one Aggregate you must make sure they are parented to one block. This is held as a typical orderstructure with the Aggregate order type (ID = 25)

An order has a one-to-many relationship with a customer request, which in turn has a one to many relationship with an aggregate order. The order to customer request relationship requires the Customer request fields defined below to be consistent between the two order types. NewAggOrderStruct is defined as:

TraderAccount	A string[21] variable containing a valid trader account for the user, as returned by *ptGetTrader*. May be set to any string that equates to a valid trader account for the logged in user.
ExchangeName	A string[11] variable containing the exchange name for the order. This must be the valid exchange name for the contract, as returned by *ptGetContract*.
ContractName	A string[11] variable containing the contract name for the order. This is one of the *ContractName* values returned by *ptGetContract*.
ContractDate	A string[51] variable containing the contract date of the contract, as returned by *ptGetContract*.
BuyOrSell	A char variable either 'B' or 'S'.

AveragePrice	String[21] variable that converts to floating point number. Not used on order entry.
Reference	A string[26] reference value that will be passed to the server along with the order and will be echoed back on
DoneForDay	A char variable that indicates the required volume is now complete. Not used on order entry. Either Y or N.
Xref	An integer variable containing a user supplied cross-reference number. This cross-reference is no longer valid if the API is exited.

The routine returns the following error codes:

ptSuccess Order has been sent to the host for processing (not the exchange).

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrUnknownAccount Trader account does not match a known trader.

ptErrUnknownOrderType Order type is not known to PATS.

ptErrUnknownContract Contract name / date does not refer to a valid contract.

ptErrTASUnavailable Transaction Server is not connected.

3.5.2 ptAddAAOrder

Arguments: PrimaryOrder struct read-only, by reference

SecondaryOrder struct read-only, by reference

BidUser struct read-only, by reference

OfferUser struct read-only, by reference

OrderIDs struct writeable, by reference

Returns: status code

The ptAddAAOrder routine submits a new off market order to the Host. These are used to report off exchange trades to the Connect hosts and consist of reporting both the buy and sell side.

PrimaryOrder Address of a structure of type NewOrderStruct containing the order details.

SecondaryOrder Address of a structure of type NewOrderStruct containing the order details.

BidUser A String[11] containing the bid user

OfferUser A String[11] containing the offer user

OrderIDs Address of a structure of type CrossingOrderIDs

NewOrderStruct is defined in *ptAddOrder*

The routine returns the following error codes:

ptSuccess Order has been sent to the host for processing (not the exchange).

ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrPriceRequired	Order type requires a price and one was not provided.
ptErrInvalidPrice	Supplied price did not convert to a valid real number
ptErrUnknownAccount	Trader account does not match a known trader.
ptErrUnknownOrderType	Order type is not known to PATS.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.
ptErrTASUnavailable	Transaction Server is not connected.
ptErrMDSUnavailable	Market Data Server is not connected.

3.5.3 ptAddBasisOrder

Arguments:	PrimaryOrder	struct	read-only, by reference
	SecondaryOrder	struct	read-only, by reference
	BasisOrder	struct	read-only, by reference
	OrderIDs	struct	writeable, by reference
Returns:	status code		

This function is a variant of the *ptAddAAOrder* routine with different set of arguments. BasisOrder is a type of BasisOrderStruct

3.5.4 ptAddBlockOrder

Arguments:	PrimaryOrder	struct	read-only, by reference
	SecondaryOrder	struct	read-only, by reference
	LegPrices	struct	read-only, by reference
	OrderIDs	struct	writeable, by reference
Returns:	status code		

This function is a variant of the *ptAddAAOrder* routine with different set of arguments. Legprices is a type of LegPriceStruct.

3.5.5 ptAddCrossingOrder

Arguments:	PrimaryOrder	struct	read-only, by reference
	SecondaryOrder	struct	read-only, by reference
	LegPrices	struct	read-only, by reference
	OrderIDs	struct	writeable, by reference
	FAK	Char	read-only, default = 'L'
Returns:	status code		

This function is a variant of the *ptAddAAOrder* routine with different set of arguments. The FAK parameter added as part of the 6.3 CME Eurodollar functionality allows the user to determine if the crossing order placed on

the exchange uses “Fill & Kill” behaviour. If the exchange supports this functionality, an order placed with the default ‘L’ setting (instructing the exchange to ‘Leave’ the unfilled leg in the market) in the FAK field will leave an unfilled leg on the exchange to be filled at a later time. If the FAK field is set to ‘K’, the unfilled leg of the cross will be pulled from the market as soon as the opposing leg is filled.
This will only occur if one of the crossed legs is filled, leaving the other in the market with some volume remaining.
LegPrices is a type of LegPriceStruct.

3.5.6 ptAddCustRequest

Arguments:	NewCustReq	struct	read-only, by reference
	OrderID	string[11]	writeable, by reference
Returns:	status code		

The ptAddCustRequest function is part of the Order Management Interface functions and allows a customer master order to be entered into the system. One or more execution orders are then placed at the exchange over a period of time to fulfill the customer’s requirements.

NewCustReq Address of a structure of type NewCustReqStruct containing the order details.

OrderID Address of a string[11] variable which will receive the OrderID of the order.

If Order Management Integeration (OMI) is enabled for the session, the user will have data structures to allow for alternative back office processing. OMI will need to be enabled on the core components, and calling ptOMIEnabled will allow the client application to determine if the OMI functionality is enabled.

Customer Request:
This contains details of a whole order such as buy 10,000 contracts at 101.04. This will be what a trader actually works and will have a price and quantity and buy/sell indicator. Many orders can be aggregated together under one Aggregate Order for allocation purposes.
An order has a one-to-many relationship with a customer request, which in turn has a one to many relationship with an aggregate order. The order to customer request relationship requires the Customer request fields defined below to be consistent between the two order types.

NewCustReqStruct is defined as:

TraderAccount	A string[21] variable containing a valid trader account for the user, as returned by *ptGetTrader*. May be set to any string that equates to a valid trader account for the logged in user.
ExchangeName	A string[11] variable containing the exchange name for the order. This must be the valid exchange name for the contract, as returned by *ptGetContract*.
ContractName	A string[11] variable containing the contract name for the order. This is one of the *ContractName* values returned by *ptGetContract*.
ContractDate	A string[51] variable containing the contract date of the contract, as returned by *ptGetContract*.

BuyOrSell	A char variable either 'B' or 'S'.
ActualAmount	The total volume required by the associated child orders to be traded
OrderType	A string[11] variable containing the order type.
Price	A string[21] variable containing the target price for the child orders. May be any real number. It is possible the child orders are filled at prices other than the one specified in the parent Customer Request order
Price 2	A string[21] variable containing the target price for the child orders. May be any real number. Used when customer request is for a stop limit order type.
ParentID	A string[11] variable that ties this information to the parent Aggregate Order. If this isn't specified, the STAS auto-generates this order
TotalVolume	An integer variable containing the number of lots placed so far by the child orders
CumulativeVol	An integer variable containing the number of lots completed for the child orders so far.
AveragePrice	A string[21] variable containing the average price of the child orders filled so far, converts to a floating point number.
Reference	A string[26] variable containing reference data attached to the order
Xref	An integer variable containing reference data to attached to the order

The routine returns the following error codes:

ptSuccess	Order has been sent to the host for processing (not the exchange).
ptErrNoPreallocOrders	API has no pre-allocated order ids.
ptErrTASUnavailable	API is not connected to the host.
ptErrMDSUnavailable	API is not connected to the MDS
ptErrUnknownAccount	Account is not known to the API
ptErrNotTradable	Account is known but not allowed to trade
ptErrInvalidOrderParent	Order ID specified is an invalid order

3.5.7 ptAddOrder

Arguments:	NewOrder	struct	read-only, by reference
	OrderID	string[11]	writeable, by reference
	Returns:	status code	

The ptAddOrder routine submits a new order to the Host. The routine does not verify that the order is sent or accepted by the exchange. This information will be available when the order state changes as indicated by the *ptOrder* callback. This function will return an error code if there is no connection to a transaction server.

The **Patsystems** order ID is automatically supplied by the host and this information will become available when the host processes the order. It will be returned by the *ptOrder* callback and it should be used to query the order details using *ptGetOrderByID*.

Synthetic orders are added using this same call, but are held locally in the API and not sent to the server until the trigger price is seen. Be aware that a disconnection of the price feed (either due to network errors or by calling *ptDisconnect*) will affect the ability to see this trigger price. However, the synthetic orders will still remain in the API, unless a logon is performed with a different username or with the reset flag set.

NewOrder Address of a structure of type NewOrderStruct containing the order details.

OrderID Address of a string[11] variable which will receive the OrderID of the order.

NewOrderStruct is defined as:

TraderAccount	A string[21] variable containing a valid trader account for the user, as returned by *ptGetTrader*. May be set to any string that equates to a valid trader account for the logged in user.
OrderType	A string[11] variable containing the order type. This is one of the values returned by *ptGetOrderType*.
ExchangeName	A string[11] variable containing the exchange name for the order. This must be the valid exchange name for the contract, as returned by *ptGetContract*.
ContractName	A string[11] variable containing the contract name for the order. This is one of the *ContractName* values returned by *ptGetContract*.
ContractDate	A string[51] variable containing the contract date of the contract, as returned by *ptGetContract*.
BuyOrSell	A char variable either 'B' or 'S'.
Price	A string[21] variable containing the target price for the order. May be any real number. If the order type does not require a price, as defined by *ptOrderTypePriceRequired* then this field must be set to blank
Price2	A string[21] variable containing a second price if required. For example, a limit price for a stop/limit order. Blank if not required.
Lots	An integer variable containing the number of lots for the order. May be any positive integer.
LinkedOrder	A string[11]. Not in current use
OpenOrClose	A char variable either 'O' or 'C' indicating if the Order is to open or close the traders position.
Xref	An integer variable containing a user supplied cross-reference number. This cross-reference is no longer valid if the API is exited.
XrefP	An integer variable containing a user supplied cross reference number. This cross-reference persists even if the API is shutdown.
GoodTillDate	A string[9] variable containing the good till date for the order as CCYYMMDD, or blank if not required. For a good till cancelled order, leave this blank.
TriggerNow	A char variable either 'Y' or 'N' indicating if synthetic orders should be checked (and triggered if necessary) immediately rather than awaiting a price update message.
Reference	A string[26] variable containing a user supplied cross reference similar in function to the XrefP but allowing text. Should be specified in addition to, not as a replacement of, XrefP.
ESARef	A string[51] variable that can be used to receive additional information from the exchange adapater. Used for FIXTGW to receive exchange order number from CME FX.

Priority	An integer variable to set priority that can be sent along with the order. Has meaning only for some exchanges.
TriggerDate	A string[9] variable containing a date to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
TriggerTime	A string[7] variable containing a time to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
BatchID	A string[11] variable used to determine orders placed/received in batches (for example, wholesale trade orders).
BatchType	A string[11] variable that describes the type of batch being placed/received (for example, a crossing batch is type 42)
BatchCount	An integer describing the number of records in the batch.
BatchStatus	A string[11] variable that describes the status of the batch as it is passed through the system.
ParentID	A string[11] variable used to determine the parent order associated with Aggregate, Customer Request, and Orders placed as part of Order Management Integeration (see ptOMIEnabled)
DoneForDay	A char variable used to determine if an Aggregate Order is completed, and if so whether the order and associated child orders can be modified. Introduced as part of the OMI functionality
BigRefField	A string[256] reference variable that will be echoed back on subsequent order responses.
SenderLocationID	A string[33]. It is geographic location of end user (2 character country code). For users based in US and Canada this should also include state code: For example, **US, IL** (for US, Illinois) **Note:** This field is required for orders placed on CME exchange. Follow this link to get list of codes ftp.cmegroup.com/fix/coo
Rawprice	A string[21]. Deprecated field previously used by FIXTGW field for CME FX.
Rawprice2	A string[21]. Deprecated field previously used by FIXTGW field for CME FX.
ExecutionID	A string[71]. Deprecated field previously used by FIXTGW field for CME FX.
ClientID	A string[21] variable used to determine the client ID of the order. For CME Globex must be populated using unique SenderSubID tag 50 value.
APIM	A Char value used to describe the APIM value required by Connect 9.0 and CME exchanges. • 'M' for manually entered orders. • 'A' for automatically entered orders.
APIMUser	A string[21] variable used to describe the ITM code assigned to the third party developer to comply with Connect 9.0 exchanges.
YDSPAudit	A string[11] value used to pass Yesterdays Settlement Price used to calculate the Near and Far leg prices when the user is trading an Inter Commodity Spread (ICS) as part of the Connect 9.0 requirements
ICSNearLegPrice	A string[11]. The calculated Near Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
ICSFarLegPrice	A string[11]. The calculated Far Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
MinClipSize	The Integer used to determine minimum clip size to be placed when a ghost order is placed. Can be used against SyOMS 2.13 and greater

MaxClipSize	The Integer used to determine minimum clip size to be placed when a ghost order is placed. Can be used against SyOMS 2.13 and greater
Randomise	A char used to determine if the Clip size is to be randomly generated when the ghost order is working
TicketType	A string[3] used to describe the type of ticket used when the ghost order was placed. Patsystemst GUI specific
TicketVersion	A String[4] used to determine the version of ticket used when twuihe ghost order was placed. Patsystemst GUI specific
ExchangeField	A String[11] - Patsystems specific field: please ignore
BOFID	A String[21] - Patsystems specific field: please ignore
Badge	A String[6] - Patsystems specific field: please ignore
LocalUserName	A String[11] - Patsystems specific field: please ignore
LocalTrader	A String[21] - Patsystems specific field: please ignore
LocalBOF	A String[21] - Patsystems specific field: please ignore
LocalOrderID	A String[11] - Patsystems specific field: please ignore
LocalExAcct	A String[11] - Patsystems specific field: please ignore
RoutingID1	A String[11] - Patsystems specific field: please ignore
RoutingID2	A String[11] - Patsystems specific field: please ignore
Inactive	Char – indicates if the order is active or not

The routine returns the following error codes:

ptSuccess	Order has been sent to the host for processing (not the exchange).
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrPriceRequired	Order type requires a price and one was not provided.
ptErrInvalidPrice	Supplied price did not convert to a valid real number
ptErrUnknownAccount	Trader account does not match a known trader.
ptErrUnknownOrderType	Order type is not known to PATS.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.
ptErrTASUnavailable	Transaction Server is not connected.
ptErrMDSUnavailable	Market Data Server is not connected.

3.5.8 ptAddAlgoOrder

Arguments:	NewOrder	struct	read-only, by reference
	BuffSize	int	read-only, by reference
	AlgoBuff	struct	read-only, by reference
	OrderID	string[11]	writeable, by reference
Returns:	status code		

The ptAddAlgoOrder routine submits a new order to the Host. It does exactly the same as ptAddOrder adding extra XML information to be used by the ALGO server. Algo Buff is defined as an array of Char.

The routine returns the following error codes:

ptSuccess	Order has been sent to the host for processing (not the exchange).
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrPriceRequired	Order type requires a price and one was not provided.
ptErrInvalidPrice	Supplied price did not convert to a valid real number
ptErrUnknownAccount	Trader account does not match a known trader.
ptErrUnknownOrderType	Order type is not known to PATS.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.
ptErrTASUnavailable	Transaction Server is not connected.
ptErrMDSUnavailable	Market Data Server is not connected.
ptErrInvalidAlgoXML	This Algo Order contains incorrect XML information

3.5.9 ptAddOrderEx

Arguments:	NewOrder	struct	read-only, by reference
	OrderID	string[11]	writeable, by reference
	UserName	string[11]	writeable, by reference
Returns:	status code		

The normal call (ptAddOrder) attaches the logon username to the trade. The ptAddOrdEx function call may be used to attach a different username to the order, for example so that a multi-user gateway application can set usernames for receiving exchange member rates on the eCBOT and CME exchanges. By specifying ptAddOrderEx and giving a different username, the exchange gateways will pick up appropriate attributes to receive the correct exchange member/non-member rates.

The function call has some restrictions placed on it. This function call must be used under the following restrictions. Failure to adhere to these restrictions may result in unexpected behaviour or revocation of your license.

1. The username must exist on the remote server
2. The application name and license details must match the logon user
3. The account for the trade must be a valid account for the attached user

NewOrder Address of a structure of type NewOrderStruct containing the order details. Refer to *ptAddOrder*.

OrderID Address of a string[11] variable which will receive the OrderID of the order.

UserName A string[11] variable to receive the username to attach to the trade.

The routine returns the following error codes:

ptSuccess	Order has been sent to the host for processing (not the exchange).
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrPriceRequired	Order type requires a price and one was not provided.
ptErrInvalidPrice	Supplied price did not convert to a valid real number
ptErrUnknownAccount	Trader account does not match a known trader.
ptErrUnknownOrderType	Order type is not known to PATS.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.
ptErrTASUnavailable	Transaction Server is not connected.
ptErrMDSUnavailable	Market Data Server is not connected.

3.5.10 ptAddProtectionOrder

Arguments:	NewOrderStruct	struct	read-only, by reference
	ProtectionStruc	struct	read-only, by reference
	OrderID	struct	writeable, by reference
Returns:	status code		

This method is used to place ‘Bracket’ orders on a Patsystems core that supports this functionality. This is a minimum 2.14 SyOMS and associated STAS as a minimum functionality.

NewOrderStruct Standard order structure defined for ptAddOrder, containing the order information for the order to be protected. This can only be a Limit Order at this time.

ProtectionStruct Contains the additional information required by SyOMS to place the orders to protect the position taken by the original order (defined in NewOrderStruct).

OrderID Order ID for the order to be protected assigned within the Trading API

ProtectionStruct is defined as:

Pr1_Price	A String[20]. The price difference between the first fill for the placed order, and the price of the first level protection order placed by SyOMS
Pr1_Volume	Integer: The total volume to be placed by SyOMS at the first protection level as the placed order is filled
Pr2_Price	A String[21]. The price difference between the first fill for the placed order, and the price of the second level protection order placed by SyOMS
Pr2_Volume	Integer: The total volume to be placed by SyOMS at the second protection level as the placed order is filled.
Pr3_Price	A String[21]. The price difference between the first fill for the placed order, and the price of the first level protection order placed by SyOMS
Pr3_Volume	Integer. The total volume to be placed by SyOMS at the third protection level as the placed order is filled.
St_Type	A String[11]. The type of synthetic Stop order to be managed by SyOMS as the placed order is filled
St_Price	A String[21]. The price difference between the first fill for the placed order, and the price of the synthetic Stop order placed by SyOMS
St_Step_1	A String[21]. Indicates the minimum price the market must move before the synthetic Trailing Stop (if used) moves
St_Step_2	A String[21]. The change in price the Trailing Stop (if used) moves when the price has moved greater than the number of steps specified in St_Step_1

The routine returns the same value as in *ptAddOrder*.

3.5.11 ptAmendOrder

Arguments:	OrderID	string[11]	read-only, by reference
	NewDetails	struct	read-only, by reference
Returns:	Status		

The ptAmendOrder routine changes the details for an order already accepted by the system. The order must be in one of the following states: *ptWorking, ptPartFilled, ptHeldOrder, ptFilled* or *ptBalCancelled*. If the order is *ptWorking, ptPartFilled* or *ptHeldOrder* then all fields must be specified in the NewDetails structure. If the order is *ptFilled* or *ptBalCancelled* then only the Trader field is valid for change and the other fields are ignored.

OrderID Address of a string[11] variable containing the **Patsystems** ID of the order to change. This value is returned to the application by the *ptOrder* callback.

NewDetails Address of a structure of type AmendOrderStruct, containing the new details for the order.

AmendOrderStruct is defined as:

Price	A string[21] variable containing the new target price for the order as a text string. May be set to any real number. If the order does not require a price, this must be set to blank.
Price2	A string[21] variable containing a second price if required. For example, a limit price for a stop/limit order. Blank if not required.
Lots	An integer variable containing the new number of lots for the order. May be set to any positive integer.
LinkedOrder	A string[11] variable containing the **Patsystems** Order ID of the order linked to this order.
OpenOrClose	A char variable either 'O' or 'C' indicating if the Order is to open or close the traders position.
Trader	A string[21] variable containing the new trader account for the order. May be any non-blank text string they equates to a valid trader for the logged on user.
Reference	A string[26] variable containing a user supplied cross reference similar in function to the XrefP but allowing text. Should be specified in addition to, not as a replacement of, XrefP. XrefP is treated as fixed for the life of the order, Reference may be altered.
Priority	An integer variable to set priority that can be sent along with the order. Has meaning only for some exchanges.
TriggerDate	A string[9] variable containing a date to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
TriggerTime	A string[7] variable containing a time to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
BatchID	A string[11] variable Used to determine orders placed/received in batches (for example, wholesale trade orders).
BatchType	A string[11] variable Describes the type of batch being placed/received (for example, a crossing batch has the batch type 42)
BatchCount	An integer describing the number of records in the batch.
BatchStatus	A string[11] variable describes the status of the batch as it is passed through the system.
ParentID	A string[11] variable used to determine the parent order associated with Aggregate, Customer Request, and Orders (see ptOMIEnabled)
DoneForDay	A char variable used to determine if an Aggregate Order is completed, and if so whether the order and associated child orders can be modified. Introduced as part of the OMI functionality
BigRefField	A string[256] reference variable that will be echoed back on subsequent order responses.
SenderLocationID	A string[33]. It is geographic location of end user (2 character country code). For users based in US and Canada this should also include state code:

	For example, **US, IL** (for US, Illinois) **Note:** This field is required for orders placed on CME exchange. Follow this link to get list of codes ftp.cmegroup.com/fix/coo
Rawprice	A string[21]. Deprecated field previously used by FIXTGW field for CME FX.
Rawprice2	A string[21]. Deprecated field previously used by FIXTGW field for CME FX.
ExecutionID	A string[71]. Deprecated field previously used by FIXTGW field for CME FX.
ClientID	A string[21] variable used to determine the client ID of the order. Can use set CME's SenderSubId tag 50 value instead if username.
ESARef	A string[51]. ESA reference.
YDSPAudit	A string[11] value used to pass Yesterdays Settlement Price used to calculate the Near and Far leg prices when the user is trading an Inter Commodity Spread (ICS) as part of the Connect 9.0 requirements
ICSNearLegPrice	A string[11]. The calculated Near Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
ICSFarLegPrice	A string[11]. The calculated Far Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
MaxClipSize	The integer used to determine the Maximum clip size to be used by SyOMS when working the Ghost order (if used) in the market
LocalUserName	A String[11] – X link referece
LocalTrader	A String[21] – X link reference
LocalBOF	A String[21] – X link reference
LocalOrderID	A String[11] – X link referece
LocalExAcct	A String[11] – X link referece
RoutingID1	A String[11] – X link referece
RoutingID2	A String[11] – X link referece
AmendOrderType	A String[11] containing the order type to be amended. Otherwise, the order type name from the order being amended.
TargetUserName	A String[11] containing the target user name

The routine returns the following error codes:

ptSuccess	Request to amend the order has been sent to the Host.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrNoData	API does not currently hold any order information.
ptErrUnknownOrder	**Patsystems** ID specified does not match a valid order.
ptErrAmendDisabled	Amend is not supported by the exchange. Use cancel/add.
ptErrInvalidState	Order may not be amended at this time.
ptErrInvalidPrice	New price is not valid.
ptErrInvalidVolume	New volume is not valid.
ptErrUnknownAccount	New trader account does not exist.

ptErrInvalidAmendOrderType	This Order cannot be amended to this Order Type

3.5.12 ptAmendAlgoOrder

Arguments:	OrderID	string[11]	read-only, by reference
	BuffSize	int	read-only, by reference
	AlgoBuffer	struct	read-only, by reference
	NewDetails	struct	read-only, by reference
Returns:	Status		

The ptAmendAlgoOrder routine changes the details for an order already accepted by the system. It does exactly the same as ptAmendOrder with the addition of Algo XML buffer to be amended. AlgoBuff is defined as an array of char.

3.5.13 ptAtBest (callback)

Arguments: AtBestUpdStruct read-only, by reference

The *ptAtBest* callback triggers when At Best price information (firm and volume) is available. The callback indicates the exchange, contract and contract-date for which there is new data. Not all exchanges support At Best prices – the Sydney Futures Exchange is one that does. At Best prices become available (if supported by the exchange) when a regular price subscription is made via *ptSubscribePrice*.

On receipt of this callback the application should call *ptGetContractAtBest* to obtain the new At Best details (firm, volume, bid or offer) and *ptGetContractAtBestPrices* to obtain the actual At Best prices.

AtBestUpdStruct is defined as:

ExchangeName:	string[11]
ContractName:	string[11]
ContractDate:	string[51]

The callback must be registered with the *ptRegisterAtBestCallback* routine.

3.5.14 ptBlankPrices

Arguments: none

The ptBlankPrices can be called by the application in response to notification, via a callback, that the application has lost connectivity with the Price Feed server.
In this circumstance, it is advisable to notify the user that all bid and offer prices can no longer be relied upon. A simple method of doing this is to remove all current prices from the screen. This routine is provided for this purpose. A subsequent call to the ptGetPrice routine will return zero for all bid and offer prices and volumes.

3.5.15 ptCancelAll

Arguments: TraderAccount string[21] read-only, by reference

Returns: Status

The ptCancelAll routine submits cancellations for all orders for the specified trader account, in any contract. Completion of the routine does not imply that the cancellations have been successful, just that the cancels have been submitted to the host. Cancels for orders nearest to the market are submitted first (by comparing limit price to current last traded price).

If the application has made a call to ptSetUserIDFilter to enable filtering, a call to ptCancelAll will only cancel orders that have been entered by the currently logged in user.

The orders may be cancelled when they are in any of the following states: *ptWorking, ptHeldOrder, ptPartFilled, ptUnconfirmedPartFilled*. Cancels will not be submitted for orders in completed states (such as filled) or in transition states (such as amend pending).

TraderAccount	Address of a string[21] variable containing the trader account to delete orders for.

The routine returns the following error codes:

ptSuccess	The cancellations have been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API holds no order information at this time.

3.5.16 ptCancelBuys

Arguments:	TraderAccount	string[21]	read-only, by reference
	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference

Returns: Status

The ptCancelBuys routine submits cancellations for all buy orders for the trader, for the Exchange-ContractName-ContractDate combination supplied. Completion of the routine does not imply that the cancellations have been successful, just that the cancels have been submitted to the host. Cancels for orders nearest to the market are submitted first (by comparing limit price to current last traded price).

If the application has made a call to ptSetUserIDFilter to enable filtering, a call to ptCancelBuys will only cancel buy orders that have been entered by the currently logged in user.

The orders may be cancelled when they are in any of the following states: *ptWorking, ptHeldOrder, ptPartFilled, ptUnconfirmedPartFilled*. Cancels will not be submitted for orders in completed states (such as filled) or in transition states (such as amend pending).

Trader	Address of a string[21] variable containing the trader account to delete orders for.
ExchangeName	Address of a string[11] variable containing the exchange name of the contract to delete orders for.
ContractName	Address of a string[11] variable containing the contract name to delete orders for.
ContractDate	Address of a string[51] variable containing the contract date to delete orders for.

The routine returns the following error codes:

ptSuccess	Cancellations have been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API holds no order information at this time.

3.5.17 ptCancelOrder

Arguments: OrderID string[11] read-only, by reference

Returns: Status

The ptCancelOrder routine submits a cancellation for the specified order. Completion of the routine does not imply that the cancellation has been successful, just that the cancel has been submitted to the host. This function will return an error code if there is no connection to a transaction server.

The orders may be cancelled when they are in any of the following states: *ptWorking, ptHeldOrder, ptPartFilled, ptUnconfirmedPartFilled*. Cancels will not be submitted for orders in completed states (such as filled) or in transition states (such as amend pending, queued).
The order will transition to *ptCancelPending* state. Further cancels for this order must not be considered unless the order reverts to one of the working states listed above. Do not submit further cancels for an order already in the pending cancel state.

OrderID	Address of a string[11] variable containing the **Patsystems** order ID of the order to be cancelled. This is the value returned by the *ptOrder* callback for a new order and uniquely identifies the order on PATS.

The routine returns the following error codes:

ptSuccess	Cancellation has been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API holds no order information at this time.
ptErrUnknownOrder	**Patsystems** order ID does not refer to a known order.
ptErrInvalidState	Order is not in a valid state to cancel.

3.5.18 ptActivateOrder

Arguments: OrderID string[11] read-only, by reference

Returns: Status

The ptActivateOrder routine submits order activation to a previous order with inactive flag set. Completion of the routine does not imply that the activation has been successful, just that the activation has been submitted to the host. This function will return an error code if there is no connection to a transaction server.

OrderID Address of a string[11] variable containing the **Patsystems** order ID of the order to be cancelled. This is the value returned by the *ptOrder* callback for a new order and uniquely identifies the order on PATS.

The routine returns the following error codes:

ptSuccess	Cancellation has been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API holds no order information at this time.
ptErrUnknownOrder	**Patsystems** order ID does not refer to a known order.
ptErrInvalidState	Order is not in a valid state to be Activated.

3.5.19 ptDeactivateOrder

Arguments: OrderID string[11] read-only, by reference

Returns: Status

The ptDeactivateOrder routine submits order deactivation to a previous order with active flag set. Completion of the routine does not imply that the deactivation has been successful, just that the deactivation has been submitted to the host. This function will return an error code if there is no connection to a transaction server.

OrderID	Address of a string[11] variable containing the **Patsystems** order ID of the order to be cancelled. This is the value returned by the *ptOrder* callback for a new order and uniquely identifies the order on PATS.

The routine returns the following error codes:

ptSuccess	Cancellation has been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	The API holds no order information at this time.
ptErrUnknownOrder	**Patsystems** order ID does not refer to a known order.
ptErrInvalidState	Order is not in a valid state to be deactivated.

3.5.20 ptCancelSells

Arguments:	TraderAccount	string[21]	read-only, by reference
	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
Returns:	Status		

The ptCancelSells routine submits cancellations for all sell orders for the trader, for the Exchange-ContractName-ContractDate combination specified. Completion of the routine does not imply that the cancellations have been successful, just that the cancels have been submitted to the host. Cancels for orders nearest to the market are submitted first (by comparing limit price to current last traded price).

If the application has made a call to ptSetUserIDFilter to enable filtering, a call to ptCancelSells will only cancel sell orders that have been entered by the currently logged in user.

The orders may be cancelled when they are in any of the following states: *ptWorking, ptHeldOrder, ptPartFilled, ptUnconfirmedPartFilled*. Cancels will not be submitted for orders in completed states (such as filled) or in transition states (such as amend pending)

Trader	Address of a string[21] variable containing the trader account to delete orders for.
ExchangeName	Address of a string[11] variable containing the exchange name of the contract to delete orders for.
ContractName	Address of a string[11] variable containing the contract name to delete orders for.

ContractDate	Address of a string[51] variable containing the contract date to delete orders for.

The routine returns the following error codes:

ptSuccess	Cancellations have been sent to the host for processing.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API holds no order information at this time.

3.5.21 ptCountFills

Arguments: Count integer writeable, by reference

Returns: Status Code

The ptCountFills function returns the number of fills held for the user in the API. This value is useful for loop control when calling ptGetFill to obtain fill details. Fills are **not** stored in the order they are received in and cannot be indexed by *index*. They are stored in Fill I.D. order.

count	Address of an integer variable where the API will write the result.

The routine returns the following error codes:

ptSuccess	Routine has completed successfully.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API holds no order information at this time.

3.5.22 ptCountOrderHistory

Arguments: Index integer read-only, immediate value

Count integer writeable, by reference

Returns: Status Code

The ptCountOrderHistory function returns the number of order history records held for the given order in the API. This value is useful for loop control when calling *ptGetOrderHistory* to obtain order history details. The order history count includes the current active (non-historical) order.

Index	Index of the order for which to retrieve the history count.
Count	Address of an integer variable where the API will write the result.

The routine returns the following error codes:

ptSuccess	Order history count was successful.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrInvalidIndex	Index value does not refer to a valid record.

ptErrNoData	API holds no order information at this time.

3.5.23 ptCountOrders

Arguments: Count integer writeable, by reference

Returns: Status Code

The ptCountOrders function returns the number of order records held for the user in the API. This value is useful for loop control when calling *ptGetOrder* to obtain order details. This value is the number of orders held in the API, with each order containing several history records that record the changes in order state.

count Address of an integer variable where the API will write the result.

The routine returns the following error codes:

ptSuccess	Order count was successful.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.

3.5.24 ptCountContractAtBest

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	Count	integer	writeable, by reference
Returns:	status		

The ptCountContractAtBest routine counts the number of At Best prices records that exist for a given contract. The returned value can be used in a loop to read the list using *ptGetContractAtBest*.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

Count Address of an integer variable in which the API will write the result.

The routine returns the following error codes:

ptSuccess	Iser is allowed access to the trader account.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownContract	Exchange, Contract and Date not recognised.

3.5.25 ptCountContractSubscriberDepth

Arguments	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	Count	integer	writeable, by reference
Returns:	status		

The ptCountContractSubscriberDepth routine counts the number of Subscriber Depth prices records that exist for a given contract. The returned value can be used in a loop to read the list using *ptGetContractSubscriberDepth*.

Subscriber Depth is similar to At Best prices, in that it provides individual volume for each firm at a given price. The routine currently only applies to the Sydney Futures Exchange.

ExchangeName	Address of a string variable containing the ASCII name of the exchange.
ContractName	Address of a string variable containing the ASCII name of the commodity.
ContractDate	Address of a string variable containing the ASCII name of the contract date.
Count	Address of an integer variable in which the API will write the result.

The routine returns the following error codes:

ptSuccess	User is allowed access to the trader account.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownContract	Exchange, Contract and Date was not recognised.

3.5.26 ptDoneForDay

Arguments: OrderID: string[11] read only, by reference

Returns: Status

The *ptDoneForDay* routine is used to indicate a particular aggregate order is now complete in the Order Management Interface functions. Refer to ptOMIEnabled.

OrderID	Address of a string[11] variable containing the order number of the curtomer request or aggregate order to be marked as done.

The routine returns the following codes:

ptSuccess	Order has been assigned a new parent

ptErrNotInitialised	API has not been initialised.
PtErrNotAggOrder	Specified order is not an Aggregate Order

3.5.27 ptFill (callback)

Arguments: OrderID string[11] read-only, by reference

FillID string[71] read-only, by reference

The ptFill callback signals that a fill has been received. Fills are usually received in response to an order, but can be received as a result of the administrator entering fill details manually (an external fill), and they can be generated by the system to show the previous nights position (a netted fill). Details are provided in the callback's OrderID parameter to identify these fill types.

OrderID Address of a string[11] variable that will contain the **Patsystems** Order ID to which the fill applies. For external or netted fills, the field will contain "EXTERNAL" or "NETTED" as appropriate.

FillID Address of a string[71] variable that will contain the **Patsystems** Fill ID of the fill. This value can be used to retrieve Fill information using ptGetFillByID.

The callback must be registered with the *ptRegisterFillCallback* routine. When a normal fill is received in response to an order trading, the *ptOrder* callback will also fire, since the order has undergone a state change. However, there is no guarantee which event will fire first, so the application must be prepared to process both.

The returned Order ID for a normal fill may be used to filter the output of *ptGetFill* so that the up to date list of all fills for the order can be read. After this callback executes, the fill list and trading position details for the trader include this latest fill.

The returned Fill ID for a normal fill can be passed to *ptGetFillByID* in order to directly access the fill details.

An EXTERNAL fill is a fill entered by the system administrator to reflect a trade or position done outside the Patsystems environment. A NETTED fill is received at the start of day to reflect an over-night position known to the Patsystems environment. The price for a NETTED fill is the settlement price from the previous day's close. These types of fills have no order ID.

3.5.28 ptGetAveragePrice

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference

	Price	string[21]	writeable, by reference
Returns	Status		

The ptGetAveragePrice routine returns the average price of the open fills for the trader in a given contract. This can be used to show how the open profit or loss fluctuates with market movement.

This function is not enabled for gateway applications. It is expected that gateway applications will remove orders and fills during processing and this invalidates the position calculation used by this routine.

ExchangeName	Address of a string[11] variable containing the exchange name of the contract to query.
ContractName	Address of a string[11] variable containing the contract name to query. This value is one of the values returned by *ptGetContract*.
ContractDate	Address of a string[51] variable containing the contract date of the contract to query. Both this and *ContractName* must be specified in the query. The value is one of the values returned by *ptGetContract*.
TraderAccount	Address of a string[21] variable containing the trader account that the query is for. Fills not for this account will be ignored.
Price	Address of a string[21] variable to contain the average price of the open fills. This value converts to a floating point number.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not hold any fill data at the current time.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.
ptErrNotEnabled	This function is not enabled for gateway applications.

3.5.29 ptGetContractAtBest

Arguments	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	Index	integer	read-only, by reference
	AtBestRec	struct	writable, by reference
Returns:	Status		

The ptGetContractAtBest routine returns the appropriate At Best data for the At Best bid and offer price for a given contract. This can be supplied if and only if the exchange supports At Best price data. Most exchange interfaces used by Patsystems do not support At Best data.

This function is designed to be used by first calling *ptCountContractAtBest* which will return the number of At Best data records for a specific Exchange-ContractName-ContractDate combination. This count data is then used to supply the upper limit for the index field in *ptGetContractAtBest* for the same Exchange-ContractName-ContractDate combination.

ExchangeName	Address of a string variable containing the ASCII name of the exchange.
ContractName	Address of a string variable containing the ASCII name of the commodity.
ContractDate	Address of a string variable containing the ASCII name of the contract date.
Index	Integer specifying which record to return. Specify a value between 0 and *ptCountContractAtBest* – 1 where the Exchange-ContractName-ContractDate combination is the same as that specified for this function.
AtBestRec	Address of a data structure of type AtBestStruct where the API will write the details. The structure is defined as:

Firm	A string[4] variable for receiving the firm related to this At Best price.
Volume	An Integer variable for receiving the volume related to this At Best price.
BestType	Contains 'B' if the At Best price is a bid or 'O' if the At Best price is an offer.

The routine returns the following error codes:

ptSuccess	Call was successful and data is returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownContract	Exchange, Contract and Date is not valid.
ptErrInvalidIndex	Value supplied for the index is out range.

3.5.30 ptGetContractAtBestPrices

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	AtBestPrices	struct	writable, by reference

Returns: Status

The ptGetContractAtBestPrices routine returns the appropriate At Best price Bid/Offer for a given contract. At Best prices are available (if supported by the exchange) once a call to *ptSubscribePrice* has been made.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

AtBestPrices Address of a data structure of type AtBestPricesStruct where the API will write the price details. The structure is defined as:

BidPrice	String[21] containing the bid price
OfferPrice	String[21] containing the offer price
LastBuyer	A String[4] containing the last buyer
LastSeller	A String[4] contanining the last seller

The routine returns the following error codes:

ptSuccess User is allowed access to the trader account.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrUnknownContract Exchange, Contract and Date was not recognised.

3.5.31 ptGetContractPosition

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	Position	struct	writeable, by reference

Returns: Status

The ptGetContractPosition routine returns the current total position of the trader for a given contract. This includes both the open and closed position. Profit is reported in contract currency. This function is not enabled for gateway applications. It is expected that gateway applications will remove orders and fills during processing and this invalidates the position calculation used by this routine.

ExchangeName Address of a string[11] variable containing the name of the exchange for the contract to be queried.

ContractName Address of a string[11] variable containing the contract name to query. This value is one of the values returned by *ptGetContract*.

ContractDate	Address of a string[51] variable containing the contract date of the contract to query. Both this and *ContractName* must be specified in the query. The value is one of the values returned by *ptGetContract*.
TraderAccount	Address of a string[21] variable containing the trader account that the query is for. Fills not for this account will be ignored.
Position	Address of a structure of type PosititionStruct where the API will write the data. PositionStruct is defined as:

Profit	A string[21] variable to contain the total profit in the contract, reported in contract currency. Converts to a floating point number.
Buy	An integer variable containing the current total buy volume in this contract.
Sell	An integer variable containing the current total sell volume in this contract.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not hold any fill data at the current time.
ptErrUnknownContract	Contract name / date does not refer to a valid contract.

3.5.32 ptGetContractSubscriberDepth

Arguments	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	Index	integer	read-only, by reference
	SubscriberDepthRec		struct writable, by reference

Returns: Status

The ptGetContractSubscriberDepth routine returns the appropriate Subscriber Depth data for a given contract. This can be supplied if and only if the exchange supports Subscriber Depth price data. Most exchange interfaces used by Patsystems do not support Subscriber Depth data, although the Sydney Futures Exchange does.
Subscriber Depth is available (when supported by the exchange) once an appropriate call to *ptSubscribeBroadcast* has been made.
This function has been designed such that it is used by first calling *ptCountContractSubscriberDepth.* This will return the number of

Subscriber Depth data records for a specific Exchange-ContractName-ContractDate combination.
The returned book can be determined by ordering the bids and offers by price.

ExchangeName	Address of a string variable containing the ASCII name of the exchange.
ContractName	Address of a string variable containing the ASCII name of the commodity.
ContractDate	Address of a string variable containing the ASCII name of the contract date.
Index	Integer specifying which record to return. Specify a value between 0 and *ptCountContractSubscriberDepth* – 1 where the Exchange-ContractName-ContractDate combination is the same as that specified for this function.
SubscriberDepthRec	Address of a data structure of type SubscriberDepthStruct where the API will write the price details. The structure is defined as:

Price	A string[21] variable for receiving the price
Volume	An Integer variable for receiving the volume
Firm	A string[4] variable for receiving the firm
DepthType	A char variable that contains B for bid or O for offer

The routine returns the following error codes:

ptSuccess	Call was successful and data is returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrUnknownContract	Exchange, Contract and Date was not recognised.
ptErrInvalidIndex	Value supplied for the index is out of the range of data.

3.5.33 ptGetFill

Arguments:	Index	integer	read-only, immediate value
	FillDetails	struct	writeable, by reference
Returns	Status		

The ptGetFill routine returns fill details, indexed by the *Index* parameter. There is no facility to filter or index data by contract or **Patsystems** order ID. The application must read the list in index order and discard any records it does not require. For example, in order to obtain all fills for an order, the entire list of fills is read and fills for other orders ignored.

Index Integer value indicating which record to return. Supply a value between 0 and ptCountFills – 1.

FillDetails Address of a structure of type FillStruct where the API will write the result. FillStruct is defined as:

Index	An integer variable containing the index number of this record. New fills may be inserted in the middle of this list. Use *ptGetFillByID* for direct access.
FillID	A string[71] variable that uniquely identifies the fill on PATS.
ExchangeName	A string[11] variable to contain the exchange name of the order.
ContractName	A string[11] variable to contain the contract name of the order.
ContractDate	A string[51] variable to contain the contract date of the order.
BuyOrSell	A char variable, either 'B' or 'S'.
Lots	An integer variable to contain the number of lots filled.
Price	A string[21] variable to contain the price the order was filled at. This value should be converted to a floating point number.
OrderID	A string[11] variable to contain the **Patsystems** ID of the order filled.
DateFilled	A string[9] variable to contain the date the fill occurred.
TimeFilled	A string[7] variable to contain the time of the fill.
DateHostRecd	A string[9] variable to contain the date the host received the fill.
TimeHostRecd	A string[7] variable to contain the time the host received the fill.
ExchOrderID	A string[31] field to contain the exchange order ID, which uniquely identifies the order on the exchange.
FillType	A byte variable to contain the fill type. This is one of: ptNormalFill, ptExternalFill or ptNettedFill. External fills are ones entered by the administrator to reflect a trade done outside the **Patsystems** environment. Netted fills appear in the morning to show the trader's overnight position.
TraderAccount	A string[21] variable to contain the trader account used to submit the order.
UserName	A string[11] variable to contain the user who submitted the order.
ExchangeFillID	String[71] Deprecated field previously used by FIXTGW field for CME FX.
ExchangeRawPrice	String[20] Deprecated field previously used by FIXTGW field for CME FX.
ExecutionID	String[71] Deprecated field previously used by FIXTGW field for CME FX.
GTStatus	Integer variable containing the Global Trtading staus of the fill
SubType	Integer variable – this is used for Settlement and Minute markets
CounterParty	String[21] Counter Party Information for SGX
Leg	String[3] A String containing the number of legs

The routine returns the following error codes:

ptSuccess Call succeeded and the data has been returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData	API does not hold any fill data at this time.
ptErrInvalidIndex	Index value specified does not refer to a valid record.

3.5.34 ptGetFillByID

Arguments:	FillID	string[71]	read-only, immediate value
	FillDetails	struct	writeable, by reference
Returns	Status		

The ptGetFillByID routine returns fill details for the fill with the given Fill ID. This provides an easy mechanism to find the fill details for a fill triggered by the *ptFill* callback. The callback will provide a Fill ID, which can be passed to this query function.

FillID Address of a string[71] variable containing the **Patsystems** Fill ID of the Fill required.

FillDetails Address of a structure of type FillStruct where the API will write the result. See *ptGetFill* for information on the FillDetailStruct structure.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not hold any fill data at this time.
ptErrInvalidIndex	The Fill ID value specified does not refer to a valid record.

3.5.35 ptGetGenericPrice

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	PriceType	integer	read-only, by reference
	Side	integer	read-only, by reference
	price	struct	writeable, by reference
Returns	Status		

The ptGetGenericPrice method allows the user to retrieve specific prices from the price structure. This is limited to the RFQ Tradable and Indicative prices at this time, and will be extended as sporadic prices of this nature are added. This will generally be called following a generic price callback received by the client application.

ExchangeName Address of a string[11] variable containing the exchange name of the contract date to be queried.

ContractName Address of a string[11] variable containing the contract name to query. This value is one of the values returned by *ptGetContract*.

ContractDate Address of a string[51] variable containing the contract date of the contract to query. Both this and *ContractName* must be specified in the query. The value is one of the values returned by *ptGetContract*.

PriceType Generic price type to be retrieved by the method

Side Integer value containing one of the following values: ptBuySide, ptSellSide, ptBothSide or ptCrossSide.

Price Price structure to be populated by the method is a PriceDetailStruct and is defined as:

Price	A string[21] variable to contain the price of the price type requested
Volume	An integer variable describing the volume (if relevant) of the price type requested
AgeCounter	Byte value, if zero, the price has expired
Direction	Byte value, 0 = same, 1 = rise, 2 = fall
Hour	Byte value
Minute	Byte value
Second	Byte value

The routine returns the following error codes:

ptSuccess Call succeeded and the data has been returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrUnknownContract Contract name / date does not refer to a valid contract.

3.5.36 ptGetOpenPosition

Arguments:

ExchangeName	string[11]	read-only, by reference
ContractName	string[11]	read-only, by reference
ContractDate	string[51]	read-only, by reference
TraderAccount	string[21]	read-only, by reference
Position	struct	writeable, by reference

Returns Status

The ptGetOpenPosition routine returns the current open position of the trader for a given contract. To evaluate this open position as the market moves, the application should call *ptGetAveragePrice* to obtain the average price of these open fills. Profit is reported in contract currency.

ExchangeName Address of a string[11] variable containing the exchange name of the contract date to be queried.

ContractName Address of a string[11] variable containing the contract name to query. This value is one of the values returned by *ptGetContract*.

ContractDate Address of a string[51] variable containing the contract date of the contract to query. Both this and *ContractName* must be specified in the query. The value is one of the values returned by *ptGetContract*.

TraderAccount Address of a string[21] variable containing the trader account that the query is for. Fills not for this account will be ignored in calculating the position.

Position Address of a structure of type PositionStruct where the API will write the data. PositionStruct is defined as:

Profit	A string[21] variable to contain the total profit in the contract, reported in contract currency. Converts to a floating point number.
Buy	An integer variable containing the current open buy volume.
Sell	An integer variable containing the current open sell volume.

The routine returns the following error codes:

ptSuccess Call succeeded and the data has been returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData The API does not hold any fill data time.

ptErrUnknownContract Contract name / date does not refer to a valid contract.

ptErrNotEnabled This function is not enabled for gateway applications.

3.5.37 ptGetOrder

Arguments: Index integer read-only, immediate value

OrderDetail struct writeable, by reference

Returns status code

The ptGetOrder routine returns the details for an order held for the user in the API. The data is returned in Order ID order.

Index An integer value indicating which record to return. Supply a value between 0 and *ptCountOrders* - 1.

OrderDetail Address of a structure of type OrderDetailStruct where the API will write the result.

OrderDetailStruct is defined as:

Index	An integer variable containing the index number of this record. This value is guaranteed to directly access the same record while the application is attached to the API.
Historic	A char variable containing Y or N. The most recent record for the order has this value set to N. All other records have this set to Y.
Checked	A char variable containing Y or N depending on whether the ptOrderChecked routine has been called yet. Example use is to record that the customer has been notified of the fill.
OrderID	A string[11] variable that identifies the order in PATS. This is the value returned by the *ptOrder* callback and used as input to other order manipulation routines. A negative number means the order is a synthetic order held locally.
DisplayID	A string[11] variable. If the order is a synthetic order not yet triggered, this field is blank. Otherwise it is the displayable version of the OrderID above.
ExchOrderID	A string[31] variable to contain the ID that uniquely identifies the order on the exchange.

UserName	A string[11] variable to contain the user who submitted the order.
TraderAccount	A string[21] variable to contain the trader account for the order.
OrderType	A string[11] variable to contain the order type of the order. This is one of the values returned by *ptGetOrderType*.
ExchangeName	A string[11] variable to contain the exchange the order was sent to.
ContractName	A string[11] variable to contain the contract name the order is for.
ContractDate	A string[51] variable to contain the contract date the order is for, Together with *ContractName* this specifies the contract that the order is in.
BuyOrSell	A char variable, either 'B' or 'S'.
Price	A string[21] variable to contain the order price. This converts to a floating point number.
Price2	A string[21] variable containing the limit price for a stop/limit order. Blank if not required.
Lots	An integer variable to contain the number of lots for the order.
LinkedOrder	A string[11] variable containing the Patsystems order Id of the other order for OCO orders.
AmountFilled	An integer variable to contain the number of lots filled so far.
NoOfFills	An integer variable containing the number of fills received so far for the order.
AveragePrice	A string[21] variable to contain the average price the order has been filled at. This value converts to a floating point number and is a decimal, even for fractionally priced contracts like the 30 Year Bond.
Status	A byte variable to contain the order status. The valid options and their meanings are listed in the OrderStatus table that follows this table
OpenOrClose	'O' if the order is opening a position, 'C' if the order is to close a position.
DateSent	A string[9] variable to contain the date the order was sent to the host as CCYYMMDD.
TimeSent	A string[7] variable containing the local time on your PC that the order was sent HHMMSS.
DateHostRecd	A string[9] variable to contain the date the host received the order as CCYYMMDD.
TimeHostRecd	A string[7] variable containing the time the host received the order. The server may run in a different time zone than your local PC.
DateExchRecd	A string[9] variable to contain the date the exchange received the order.
TimeExchRecd	A string[7] variable containing the time the exchange received the order. The server may run in a different time zone than your local PC.
DateExchAckn	A string[9] variable to contain the date the exchange acknowledged receipt of the order.
TimeExchAckn	A string[7[variable to contain the time the exchange acknowledged the order. The server may run in a different time zone than your local PC.
NonExecReason	A string[61] variable containing the reason the order was not executed or other text information.
Xref	An integer variable containing the user supplied cross reference tag set in *ptAddOrder*. Is zero if the API has been such down since the record was added.
XrefP	An integer variable containing the user supplied cross reference tag set in *ptAddOrder*. This cross reference persists even if the API has been shutdown

UpdateSeq	An integer variable containing the sequence in which the historical records should be read, from lowest (earliest) to highest (latest).
GoodTillDate	A string[9] variable containing the good till date for the order as CCYYMMDD.
Reference	A string[26] variable containing an extended Pats order reference.
Priority	An integer variable to set priority that can be sent along with the order. Has meaning only for some exchanges.
TriggerDate	A string[9] variable containing a date to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
TriggerTime	A string[7] variable containing a time to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
Sub-state	An integer variable reflecting sub-states of main order states. Generally with is used to distinguish orders working at the exchange and those working on the SyOMS server.
BatchID	A string[11] variable Used to determine orders placed/received in batches (for example, wholesale trade orders).
BatchType	A string[11] variable Describes the type of batch being placed/received (for example, a crossing batch has the batch type 42)
BatchCount	An integer describing the number of records in the batch.
BatchStatus	A string[11] variable describes the status of the batch as it is passed through the system.
ParentID	A string[11] variable used to determine the parent order associated with Aggregate, Customer Request, and Orders placed as part of Order Management Integeration (see ptOMIEnabled)
DoneForDay	A char variable used to determine if an Aggregate Order is completed, and if so whether the order and associated child orders can be modified. Introduced as part of the OMI functionality
BigRefField	A string[256] reference variable that will be echoed back on subsequent order responses.
Timeout	Integer used by RFQT orders, this is returned by the exchange with a timeout duration for which the RFQT is valid. Only available if the RFQT flag is enabled for the exchange.
QuoteID	String[121] Used by RFQT orders, this is returned by the exchange with a timeout duration for which the RFQT is valid. When an update is received from the exchange regarding this RFQ with a new time, this Quote ID is used to identify the RFQ in question (as this could be relevant across different placed RFQ orders with different PATS order ids). Only available if the RFQT flag is enabled for the exchange.
LotsPosted	Integer value – not in use
ChildCount	Integer value – not in use
ActLots	Integer value – not in use
SenderLocationID	A string[33]. It is geographic location of end user (2 character country code). For users based in US and Canada this should also include state code: For example, **US, IL** (for US, Illinois) **Note:** This field is required for orders placed on CME exchange. Follow this link to get list of codes ftp.cmegroup.com/fix/coo
Rawprice	String[21] Deprecated field previously used by FIXTGW field for CME FX.
Rawprice2	String[21] Deprecated field previously used by FIXTGW field for CME FX.
ExecutionID	String[71] Deprecated field previously used by FIXTGW field for CME FX.

ClientID	A string[21] variable used to determine the client ID of the order. Can be used to set CME's SenderSubId tag 50 value instead of username.
ESARef	String[51] Deprecated field previously used by FIXTGW field for CME FX.
ISINCode	A string[21] variable used to represent the ISIN Code entered when a Basis order is placed
CashPrice	A string[21] variable used to hold the cash price entered when a Basis order is placed
Methodology	A char variable used to represent the value entered for the Methodology when a Basis order is placed
BasisRef	A string[21] variable used to determine any reference passed back to the client for a Basis order
EntryDate	A string[9] value used to hold the date the order was sent from the STAS when the order was originally placed
EntryTime	A string[7] value used to hold the time the order was sent from the STAS when the order was originally placed
APIM	A Char value used to describe the APIM value required by Connect 9.0 and CME exchanges. • 'M' for manually entered orders. • 'A' for automatically entered orders.
APIMUser	A string[21] variable used to describe the ITM code assigned to the third party developer to comply with Connect 9.0 exchanges.
ICSNearLegPrice	A string[11] used to hold the calculated Near Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
ICSFarLegPrice	A string[11] used to hold the calculated Far Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
CreationDate	Placed by the API, this is a string[9] used to determine the date the order was created originally.
OrderHistorySeq	The integer used to determine the sequence of the order update in the lifcycle of the order
MinClipSize	The integer used to determine the minimum clip size to be placed for a ghost order
MaxClipSize	The integer used to determine the mazimum clip size to be placed for a ghost order
Randomise	The Char used to determine if the clips placed by a ghost order are random.
ProfitLevel	The Char used to determine the protection level of the order if it is an order placed by SyOMS as part of a bracket order.
OFSeqNumber	Integer variable – Patsystems specific field
ExchangeField	String[11] variable - Patsystems specific field
BofID	String[21] variable - Patsystems specific field
Badge	String[6] - Patsystems specific field
GTStatus	Integer variable containing the Global Trading status of the order
LocalUserName	String[11] variable used in Cross link
LocalTrader	String[21] variable used in Cross link
LocalBOF	String[21] variable used in Cross link
LocalOrderID	String[11] variable used in Cross link
LocalExAcct	String[11] variable used in Cross link

RoutingID1	String[11] variable used in Cross link
RoutingID2	String[11] variable used in Cross link
FreeTextField1	String[21] free text field
FreeTextField2	String[21] free text field
Inactive	Char indicating if the order is inactive or not

Order States

PtQueued	Submitted to PATSAPI
PtSent	Received by **Patsystems** server, order is in transit
PtWorking	Accepted by exchange as a valid order
PtRejected	Rejected, either by **Patsystems** or by the exchange
PtPartFilled	Order has been partly filled
PtFilled	Order has been completely filled
PtCancelled	Order has been cancelled
ptBalCancelled	The outstanding balance has been cancelled
ptCancelPending	The requested cancel received by **Patsystems** server, order is in transit
ptAmendPending	The requested amend received by **Patsystems** server, order is in transit
ptUnconfirmedFilled	The order has filled but the fills have not yet reached PATS
ptUnconfirmedPartFilled	The order has part filled but the fills have not reached PATS
ptHeldOrder	Order is a synthetic order waiting for price to trigger
ptCancelHeldOrder	Synthetic order has been cancelled
ptTransferred	Transferred the order to a trader account not in the user's Trader Account Group
ptExternalCancelled	The order was cancelled because the exchange has closed

Unconfirmed fills: The Unconfirmed states may result from an inquiry on the exchange (made by *ptQueryOrderStatus*), an order amendment or cancellation. These may notify the system that volume has executed but the resulting fill has not been received. This state can be turned off by specifying the application as type *ptGateway*

Held orders: These order states apply to orders held locally in the API, not on the server.

3.6 Order Sub-States

ptSubStatePending	The order is being held by SYOMS, prior to being triggered by market conditions
ptSubStateTriggered	The synthetic order has been triggered and is working in the market

The routine returns the following error codes:

ptSuccess	Call succeeded, the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the host.
ptErrNoData	API does not hold any data at this time.
ptErrInvalidIndex	Index value does not refer to a valid record.

3.7 Fill Sub-Types

ptFillSubTypeSettlement	A settlement market fill
ptFillSubTypeMinute	A minute market fill
ptFillSubTypeUnderlying	A fill message for the underlying leg when the market closes for minute or settlement order
ptFillSubTypeReverse	A reverse fill – always has a negative volume to cancel the orignal position on the minute or settlement market order

3.8 Global Trading States

During the Global Trading session (24 hours a day, 6 days a week) exchanges will open and close, contract dates will expire and session data will be updated. For this reason there are additional states for Exchanges, Commodities, Contracts, Orders and Fills:

ptGTUndefined = 0;	Undefined Global Trading state
ptGTActive = 1;	Object is within open hours for the exchange
ptGTInactive = 2;	Object is out of open hours for the exchange
ptGTExpired = 3;	Object has expired and is available for purging

3.8.1 ptGetOrderEx

Arguments:	Index	nteger	read-only, immediate value
	AlgoDetail	struct	writable, by reference
	AlgoSize	int	writable, by reference
	OrderDetail	struct	writeable, by reference
Returns:	status code		

The ptGetOrderEx routine returns the details for an order held for the user in the API. It does exactly the same as ptGetOrder with the addition Algo XML structure and size. The return values are the same for ptGetOrderEx.

AlgoDetail is defined as an array of char.

3.8.2 ptGetOrderByID

Arguments:	OrderID	string[11]	read-only, by reference
	OrderDetail	struct	writeable, by reference
	OFSequence	Int	read-only, immediate value
Returns:	Status code		

The ptGetOrderbyId routine returns the details for an order held for the user in the API. The data is returned in chronological order.

OrderID Address of a string[11] variable containing the **Patsystems** order ID of the order to query. This value is returned by the *ptOrder* callback and uniquely identifies the order on PATS. Synthetic orders managed by the API have Ids starting eith “S”.

OrderDetail Address of a structure of type OrderDetailStruct where the API will write the result. See *ptGetOrder* for details of OrderDetailStruct.

OFSequence The index of the order update within the list of order updates for that order ID, where 1 is the first order update. Value is defaulted to zero for backwards compatibility. This value is passed back to the client in the Order Callback.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the host.
ptErrNoData	API does not hold any order information at this time.
ptErrUnknownOrder	**Patsystems** order ID did not refer to a valid order.

3.8.3 ptGetOrderByIDEx

Arguments:	OrderID	string[11]	read-only, by reference
	OrderDetail	struct	writeable, by reference

	AlgoDetail	struct	writable, by reference
	AlgoSize	Int	writable, by reference
	OFSequence	Int	read-only, immediate value
Returns:	Status code		

The ptGetOrderbyId routine returns the details for an order held for the user in the API. It does exactly the same as ptGetOrderById adding the extra Algo XML information and buffer size. The return values are also the same as ptGetOrderById.

AlgoDetail is defined as an array of char.

3.8.4 ptGetOrderHistory

Arguments:	Index	integer	read-only, immediate value
	Position	integer	read-only, immediate value
	OrderDetail	struct	writeable, by reference
Returns:	Status code		

The ptGetOrderHistory routine returns the details for a version of an order held for the user in the API. The data is returned in reverse chronological order (ie. newest first).

Index	Index of the order for which to retrieve the history record.
Position	Position of the history record.
OrderDetail	Address of a structure of type OrderDetailStruct where the API will write the result. See *ptGetOrder* for a complete description of OrderDetailStruct.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the host.
ptErrNoData	API does not hold any order information at this time.
ptErrInvalidIndex	Index or position value does not refer to a valid record.

3.8.5 ptGetOrderHistoryEx

Arguments:	Index	integer	read-only, immediate value
	Position	integer	read-only, immediate value
	OrderDetail	struct	writeable, by reference
	AlgoDetail	struct	writable, by reference
	AlgoSize	int	writable, by reference

Returns: Status code

The ptGetOrderHistoryEx routine returns the details for a version of an order held for the user in the API. It does exactly as ptGetOrderHistory with the additional Algo XML buffer and buffer size. The return is also the same as ptGetOrderHistory. AlgoDetail is defined as an array of char.

3.8.6 ptGetPrice

Arguments: Index integer read-only, immediate value

CurrentPrice struct writeable, by reference

The ptGetPrice routine returns price information for a contract, indexed by the *Index* parameter. This index matches the index used by *ptGetContract* on a one-for-one basis. Prices are returned in a record structure. Each "price" contains the price, the volume and an age counter. The volume field does not apply to all fields.

Index Integer specifying which record to return. Specify a value between 0 and *ptCountContracts* – 1. The index matches the index for this other routine on a one-for-one basis, so that *ptGetPrice(n)* returns prices for the contract returned by *ptGetContract(n)*.

CurrentPrice Address of a structure of type PricesStruct where the current prices will be written. PriceStruct is a structure containing multiple occurrences of a structure PriceDetailsStruct, one for each price, followed by the contract date market status and a mask indicating the prices which have changed since the last call to ptGetPrice or ptGetPriceForContract.

3.9 PriceDetailStruct

Price	A string[21] variable containing the price. Converts to a floating point number. The price applies to all price types other than "Total"
Volume	An integer variable containing the volume. volume applies to "Bid", "Offer", "Last", "Total" and all "DOM" fields. For other price types this field will be zero.
AgeCounter	A byte variable containing the value of the price countdown timer. If it is set to *MaxAge* then this is a fresh price, if zero then the counter has expired.
Direction	A byte variable indicating the direction of price movement since the last price. This will be *ptPriceNormal*, *ptPriceRise* or *ptPriceFall*.
Hour	A byte variable containing the hour that the price was received.
Minute	A byte variable containing the minute that the price was received.
Second	A byte variable containing the second that the time was received.

The *AgeCounter* value can be used to determine whether a price type has been updated or expired. It is maintained for all price items including opening, closing, depth and time & sales last traded prices.

It must be realised that shortly after some of these prices are received (for example, the intra-day high) they may expire and the age counter become zero. This is normal if these prices are updated at a low frequency and does not indicate a fault.

If a Bid, Offer or Last traded price expires then this should be noted: these prices are updated on a frequent basis and should not expire in a busy market.

The Limit Up, Limit Down, Execution Up, Execution Down and Reference Price describe elements of specific exchanges. The Limit prices describe the price range available for the contract during the day. The Execution prices describe the current price range that can be traded, and the Reference price (usually the last traded price) describes the mid-point of the execution prices.

The settlement prices are now split between Current Settlement Price (pvCurrStl, and equates to the settlement price received at the end of the previous days trading), SOD Settlement (pvSODSTL, and describes the Settlement Price received from the exchange at the beginning of the trading day), YDSP(pvYDStl, and describes the yesterday settlement) and New Settlement Price (pvNewStl, and describes a settlement price received during the trading day).

Indicative Bid and Indicative offer determine any indicative prices broadcast by the exchange for the contracts in question.

3.10 PriceStruct

There are 20 levels of *Depth of Market* data but not all are always filled in. There are 20 levels of *Last Traded* data for providing time and sales.

Field Name	Field Type	Description
Bid	PriceDetailStruct	Best Bid
Offer	PriceDetailStruct	Best Offer
ImpliedBid	PriceDetailStruct	Implied Bid
ImpliedOffer	PriceDetailStruct	Implied Offer
RFQ	PriceDetailStruct	RFQ, Request For Quote
Last0	PriceDetailStruct	Last Traded [0..20], 0 is the most recent
…etc…		
Last19	PriceDetailStruct	Last 20 Trades
Total	PriceDetailStruct	Total Traded Volume
High	PriceDetailStruct	High
Low	PriceDetailStruct	Low
Opening	PriceDetailStruct	Opening
Closing	PriceDetailStruct	Closing
BidDOM0	PriceDetailStruct	Bid Level 0 (Depth of market)
…etc…		

Field Name	Field Type	Description
BidDOM19	PriceDetailStruct	Bid Level 19 (Depth of market)
OfferDOM0	PriceDetailStruct	Offer Level 0 (Depth of market)
…etc…		
OfferDOM19	PriceDetailStruct	Offer Level 19 (Depth of market)
LimitUp	PriceDetailStruct	Limit Up
LimitDown	PriceDetailStruct	Limit Down
ExecutionUp	PriceDetailStruct	Execution Up
ExecutionDown	PriceDetailStruct	Execution Down
ReferencePrice	PriceDetailStruct	Reference Price (relevant for TGE exchange only)
pvCurrStl **(Legacy – no longer in use)**	PriceDetailStruct	Current Settlement Price
pvSODStl **(Legacy – no longer in use)**	PriceDetailStruct	Settlement Price received from the exchange at the beginning of the trading day.
pvNewStl **(Legacy – no longer in use)**	PriceDetailStruct	Settlement Price received during the trading day.
pvIndBid	PriceDetailStruct	Indicativer Bid Price (relevant for CME exchange only).
pvIndOffer	PriceDetailStruct	Indicative Offer Price (relevant for CME exchange only).
Status	32 bit Integer	An integer containing the current Market Status of the contract date. See *ptStatusChange* for details.
Mask	32 bit Integer	An integer bitmask indicating which prices have changed since the last call to ptGetPrice or ptGetPriceForContract.
PriceStatus	32 bit Integer	Price status (relevant for TGE exchange only).

The value for the Mask in the PriceStruct is a set of bits where each bit position marks that a particular type of price changed. To test for specific price changes, perform a logical AND operation against the following enumerated types supplied by the API.

ptChangeBid	Bid Price/Volume has change
ptChangeOffer	Offer Price/Volume has changed
ptChangeImpliedBid	Implied Bid Price/Volume has changed
ptChangeImpliedOffer	Implied Offer Price/Volume has changed
ptChangeRFQ	RFQ volume has changed
ptChangeLast	Last 20 prices have changed
ptChangeTotal	Total Traded Volume has changed
ptChangeHigh	High Price has changed

ptChangeLow	Low Price has changed
ptChangeOpening	Opening Price has changed
ptChangeClosing	Closing (Settlement) Price has changed
ptChangeBidDOM	Bid DOM Prices have changed
ptChangeOfferDOM	Offer DOM Prices have changed

The routine returns the following error codes:

ptSuccess	Call succeeded. The data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not hold any data at this time.
ptErrInvalidIndex	Index does not refer to a valid record.

3.10.1 ptGetPriceForContract

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	CurrentPrice	struct	writeable, by reference

The ptGetPriceForContract routine returns price information for a contract. This call is similar in nature to ptGetPrice.

ExchangeName	Address of a string[11] variable containing the exchange name for the contract date to be queried.
ContractName	Address of a string[11] variable that contains the contract name to query.
ContractDate	Address of a string[51] variable that contains the contract date of the contract to query.
CurrentPrice	Address of a structure of type PriceStruct where the current prices will be written. See *ptGetPrice* for information on PriceStruct.

The routine returns the following error codes:

ptSuccess	Call succeeded and the data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API has not logged on to the host.
ptErrNoData	API does not currently hold any fill data.
ptErrInvalidIndex	Index does not refer to a valid record.

3.10.2 ptGetTotalPosition

Arguments: TraderAccount string[21] read-only, by reference
Position struct writeable, by reference

Returns Status

The ptGetTotalPosition routine returns the current total overall position of the trader over all contracts. This includes the open and closed position. Profit is reported in the system currency.
This function is not enabled for gateway applications. It is expected that gateway applications will remove orders and fills during processing and this invalidates the position calculation used by this routine.

TraderAccount Address of a string[21] variable containing the trader account that the query is for. Fills not for this account will be ignored in calculating the position.

Position Address of a structure of type PositionStruct where the API will write the data. PositionStruct is defined as:

Profit	A string[21] variable to contain the total profit for the trader account, reported in the system currency. Converts to a floating point number.
Buy	An integer variable containing the current total buy volume.
Sell	An integer variable containing the current total sell volume.

The routine returns the following error codes:

ptSuccess Call succeeded, the data has been returned.

ptErrNotInitialised API has not been initialised.

ptErrNotLoggedOn API has not logged on to the host.

ptErrNoData API doesot hold any data at the this time.

3.10.3 ptOrder (callback)

Arguments: Data struct writeable, by reference

The ptOrder callback signals that an order has undergone a status change. The callback returns a structure containing the identity of the order that has changed, and its old Order ID. It does not contain information about the change itself. The calling program should then use the *ptGetOrderByID* function to obtain the latest details of the order.

Data Address of a structure of type OrderUpdStruct containing details about what order changed. OrderUpdStruct is defined as:

OrderID	A string[11] variable containing the order ID.
OldOrderID	A string[11] variable containing the original order ID. This value is used when the order ID changes from the local number to the order ID assigned by the server. For example, N1 to 100010 when order goes from Queued to Sent.
OrderStatus	A byte variable containing the current status of the order.
OFSeqNumber	Integer value containing the order sequence number
OrderTypeID	Integer containing the order type ID. Future reference

Orders managed on the server, including SyOMS order types, will show the PATS Order ID immediately, because the API pre-allocates order numbers on the server.
Locally managed synthetic orders will reflect the change in order number from temporary number (such as S1) to a PATS Order ID (such as 100123) for synthetic orders. This occurs when the trigger price is reached. Be aware that disconnection of the price feed will affect the ability of the API to see the trigger price.
The callback must be registered with the *ptRegisterOrderCallback* routine.

3.10.4 ptOrderChecked

Arguments: OrderID string[11] read-only, by reference
Checked char read-only

The ptOrderChecked function sets the *Checked* field for the order. This field is available to read in the *OrderDetailStruct* structure returned by *ptGetOrder*. The *Checked* field has no effect on the API in any way – its use is designed to be purely external to the API. The API preserves the value over shutting down and restarting of the application.
An example use of the field and this function would be to maintain a reconciliation flag to indicate that the electronic order has been checked against the corresponding paperwork, or that the customer has been notified of his fill.
The function returns the following error codes:

ptSuccess	The field has been updated
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API is not logged on to the Host.
ptErrNoData	The API has no data at this time
ptErrUnknownOrder	The order specified does not exist

3.10.5 ptPriceSnapshot

Arguments: ExchangeName string[11] read-only, by reference
ContractName string[11] read-only, by reference

	ContractDate	string[51]	read-only, by reference
	Wait	Integer	read-only, by reference

Returns: Status

The ptPriceSnapshot routine requests the Price Server to supply prices for the instrument passed to it. The routine can either wait for a reply from the price server, or return immediately. In either case the receipt of the prices will be notified by the *ptPriceUpdate* callback routine.

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

Wait An integer value containing the number of milliseconds to wait for the price reply. If this value is set to zero, the routine will return immediately with ptSuccess. If the value is set to INFINITE ($FFFFFFFF), the routine will wait indefinitely for a price reply. For any other value the routine will wait for the specified amount of time. If a price reply occurs before the time has run out, ptSuccess will be returned. If the routine times out, ptErrFalse will be returned.

The routine returns the following error codes:

ptSuccess When Wait is zero, indicates that the request was sent to the price server. When Wait is non-zero, indicates that a reply was received before the time ran out.

ptErrFalse “Wait” milliseconds has elapsed and no price reply has been received.

ptErrUnknownContract The Exchange, Contract and Date was not recognised.

ptErrNotInitialised API has not been initialised with *ptInitialise*.

ptErrNotLoggedOn API is not currently logged on to the host.

3.10.6 ptPriceStep

Arguments:	Price	double	read-only, by reference
	TickSize	double	read-only, by reference
	NumSteps	Integer	read-only, by reference
	TicksPerPoint	Integer	read-only, by reference

Returns: double, adjusted price

The ptPriceStep routine can be used to adjust a price up or down by a number of ticks.

Price Original price to adjust from

TickSize Minimum tick size for the commodity to which the price belongs.

NumSteps Number of steps by which to adjust the price. Can be negative.

TicksPerPoint Number of ticks per point for the commodity to which the price belongs.

The routine is used to correctly tick up or down a price by a certain number of ticks. This means that fractional price movements can be correctly calculated.

3.10.7 ptPriceUpdate (callback)

Arguments: Data struct writeable, by reference

The ptPriceUpdate callback fires whenever a new price is received for any contract. The callback returns the contract for which the price has changed.

Data Address of a structure of type PriceUpdStruct containing details about what price changed. PriceUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.

The routine must be registered with the *ptRegisterPriceCallback* routine.

3.10.8 ptPurge

Arguments	PDate	String	read only, by reference
	PTime	String	read only, by reference
Returns	none		

ptPurge is called to purge expired objects from memory within the API. Objects are expired when a contract date expires and all its orders and fills are automatically expired. These objects will persist within the API until ptPurge is called or the user logs out.
ptPurge takes 2 arguments: the date and time purge is performed by the client application, taken directly from the system time. These parameters are to remove the situation where the API receives extra expired updates for objects that the client has not yet received and therefore purges more objects than the client during the purge.
ptPurge does not return a value but it does trigger the purgeComplete callback on each exchange as the purge is completed for each exchange.

3.10.9 ptQueryOrderStatus

Arguments: OrderID: string[11] read-only, by reference

Returns: Status

Some exchanges, such as Eurex, may sometimes delay the delivery of a fill. Such exchanges provide a mechanism where we can query the order status and even if the fill details are not available, our servers can determine the quantity of the order that has been filled.
The ptQueryOrderStatus routine issues such a message to the host. If a fill has in fact been delayed, this action may result in an order state change to one of the "Unconfirmed" states. For example, the state may change from "Working" to "Unconfirmed Filled".
The call itself does not provide any order status information. This will be returned using the normal *ptOrder* callback mechanism.

OrderID Address of a string[11] variable containing the **Patsystems** Order ID that identifies this order on the system.

The function returns the following error codes:

ptSuccess	Request to query the order has been sent
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrNoData	API does not currently hold data
ptErrUnknownOrder	ID specified does not match a valid order.
ptErrQueryDisabled	Order query is not supported by the exchange.
ptErrInvalidState	Order may not be amended at this time.

3.10.10 ptReParent

Arguments: OrderID: string[11] readonly, by reference
DestParentID string[11] readonly, by reference

Returns: Status

The *ptReParent* routine is used to re-assign an execution order to a different ustomer request order, or Customer request order to an aggregate order in the Order Management Interface functions.

OrderID Address of a string[11] variable containing the order number to be transferred to the new parent.

DestParentID Address of a string[11] variable containing the new parent order number that this order will belong to.

The routine returns the following codes:

ptSuccess	Order has been assigned a new parent
ptErrNotInitialised	API has not been initialised.

3.10.11 ptSetUserIDFilter

Arguments: Enable: char readonly, immediate value

Returns: Status

The *ptSetUserIDFilter* routine is used to enable or disable the filtering of order records. This will be applied when cancelling multiple orders with the *ptCancelAll, ptCancelBuys* and *ptCancelSells.*
It is possible to configure several user logons to have access to the same account and each logon will see all trades for the account, including any entered by the other user. Calling this function will alter the *ptCancelAll, ptCancelBuys* and *ptCancelSells* functions to apply only to trades entered by the currently logged in user rather than all trades for the account.

Enable A character 'Y' will enable filtering of orders by the currently logged in user ID, altering the behaviour of the cancellation routines listed above. Specifying 'N' disables this filtering. By default, filtering is disabled.

The routine returns the following codes:

ptSuccess	Routine has set the user ID filter on or off as requested.
ptErrNotInitialised	API has not been initialised.

3.10.12 ptSnapdragonEnabled

This function is deprecated and no longer has meaning.

3.10.13 ptStatusChange (callback)

Arguments: Data struct writeable, by reference

The ptStatusChange callback fires whenever a contract date's market status changes. The callback returns the contract that the status applies to and also the new status.

Data Address of a structure of type StatusUpdStruct containing details about what status changed, and its new value.

StatusUpdStruct is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.
Status	An integer bitmask to define what market status changes have occurred for this contract maturity.

The Status value contains a bitmask where each bit represents a status change. To test for a particular status value you should perform a logical

AND operation between the Status value and the enumerated types listed below and supplied by the API.
Not all state changes apply to all markets. Many markets do not report state changes through the **Patsystems** servers at all.

ptStateExDiv	Ex-dividend status
ptStateAuction	Auction status
ptStateSuspended	Suspended status
ptStateClosed	Closed status
ptStatePreOpen	Pre-Open status
ptStatePreClose	Pre-Close status
ptStateFastMarket	Fast Market Status

The routine must be registered with the *ptRegisterStatusCallback* routine.

3.10.14 ptSubscriberDepthUpdate (callback)

Arguments: SubscriberDepthUpdateStruct read-only, by reference

ExchangeName	string[11]
ContractName:	string[11]
ContractDate	string[51]

The *ptSubscriberDepthUpdate* callback is triggered whenever the Subscriber Depth data has changed for a contract. The callback will notify what exchange, contract and contract-date has had an update. The application should then call *ptGetContractSubscriberDepth* to obtain the price, firm and volume information.
Most exchanges do not supply subscriber price details. The Sydney Futures Exchange is one exchange that does. Subscriber Depth data is available (if supported by the exchange) once a call to *ptSubscribeBroadcast* has been made.

3.10.15 ptTicker (callback)

Arguments: Data struct writeable, by reference
Patsystems is not designed as a tick-by-tick price feed, but when used in conjunction with a real time price feed can provide a close approximation of a ticker.
The ticker callback fires whenever a bid, offer or last price or volume has altered. The intention is to provide a ticker feed of prices, such that no prices are missed. This may cause the prices to be delivered more slowly that the regular price callback.
The ticker callback must be registered using the ptRegisterTickerCallback function.
The regular price callback receives prices every 100 milliseconds or so – the regular Market Data Server issues price updates to regular API connections no more often than every 100ms. Registering the ticker callback will inform the Market Data Server to send prices in a close to real time manner – there is still a small timing window.

The regular price callback simply notifies your application that a price has updated and the application must then query for the price. This introduces yet another timing window during which price updates may be missed. The regular interface is suitable for a trading display of quotes that is being viewed by a user. Due to the timing windows mentioned, it is not suitable for a program that depends on a ticker interface to capture every trade.
The ticker callback contains the actual price updated in the returned data structure.

This data structure is defined as:

ExchangeName	A string[11] variable containing the exchange name.
ContractName	A string[11] variable containing the contract name.
ContractDate	A string[51] variable containing the contract date.
BidPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
BidVolume	An integer variable containing the volume.
OfferPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
OfferVolume	An integer variable containing the volume.
LastTradedPrice	A string[21] variable containing the price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CBOT products.
LastTradedVolume	An integer variable containing the volume.
Bid	A char variable, either Y or N, to indicate if this message contains an update to the bid or bid volume.
Offer	A char variable, either Y or N, to indicate if this message contains an update to the offer or offer volume.
Last	A char variable, either Y or N, to indicate if this message contains an update to the last or last volume.

Note that in order to get every last trade, you may also need to query the current total traded volume in order to determine if two identical last trade price and volume messages are as a result of two trades or a result of one trade and a bid or offer being pulled from the market.

3.11 Buying Power Functions

The following functions obtain buying power details from the API. Buying Power (or “Cash Margining” as it is more correctly known) is an alternative means of risk management, using available Net Liquidity to the trader, the trader’s Profit & Loss, and the Margin Per Lot required when trading a specific contract. Buying power risk management is applied per trader account.
There are several terms that go into calculating Cash Margining.

Margin Required

The Margin For Trade of any magnitude is defined as
MarginReqd = MPL*Vol
Where

MPL is the margin-per-lot required to trade the specific contract

Vol is the lot size of the order

Open Position Exposure

This value is accumulated each time a trade is made that creates or increases an open position. This is regardless of whether the open position is long or short. Working orders are also taken into account to calculate their potential impact on the open position via a worst-case scenario. The resulting figure is an integer representing the Open Position Exposure for the trader in a given contract.

Buying Power Remaining

Buying Power remaining is expressed as:
BPremain = SODLNV – OPE + P&L
Where

SODLNV	is the start of day net liquidity value, loaded from the backoffice
OPE	is the open position exposure
P&L	is the current total profit and loss

The following routines will often return data as a percentage. In these cases, the values are expressed as a percentage of the available buying power, defined as SODLNV+P&L.

3.11.1 ptBuyingPowerRemaining

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	BPRemaining	string[21]	writeable, by reference
Returns:	Status		

ExchangeName	Address of a string variable containing the ASCII name of the exchange.
ContractName	Address of a string variable containing the ASCII name of the commodity.
ContractDate	Address of a string variable containing the ASCII name of the contract date.
TraderAccount	Address of a string variable containing the ASCII name of the trader account.
BPRemaining	Address of a string variable containing the ASCII value for the Buying Power Remaining as a percentage of the SODLNV+P&L.

The ptBuyingPowerRemaining routine is used to retrieve the buying power remaining for a trader account for a given contract. If an invalid contract is passed, then the total buying power remaining for the given trader account is returned.
The function returns the following error codes:

ptSuccess	Call was successful and data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrUnknownAccount	Supplied TraderAccount name does not refer to a valid record.

3.11.2 ptBuyingPowerUsed

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	BPUsed	string[21]	writeable, by reference
Returns:	Status		

The ptBuyingPowerUsed routine is used to retrieve the buying power used for a trader account for a given contract. If an invalid contract is passed, then the total buying power used for the given trader account is returned.
The routine expects the following arguments:

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate Address of a string variable containing the ASCII name of the contract date.

TraderAccount Address of a string variable containing the ASCII name of the trader account.

BPUsed Address of a string variable containing the ASCII value for the Buying Power Used as a percentage of SODLNV+P&L.

The function returns the following error codes:

ptSuccess	Call was successful and data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrUnknownAccount	Supplied TraderAccount name does not refer to a valid record.

3.11.3 ptMarginForTrade

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	Lots	Integer	read-only, by reference
	OrderID	String[11]	read-only, by reference
	Price	String[21]	read-only, by reference
	MarginReqd	string[21]	writeable, by reference
Returns:	Status		

The *ptMarginForTrade* routine is used to retrieve the margin for a trade about to take place, for a trader account for a given contract. This retuns the current margin requirement for this trade and for maintaining any existing positions.
The routine expects the following parameters:

ExchangeName Address of a string variable containing the ASCII name of the exchange.

ContractName Address of a string variable containing the ASCII name of the commodity.

ContractDate	Address of a string variable containing the ASCII name of the contract date.
TraderAccount	Address of a string variable containing the ASCII name of the trader account.
Lots	Address of an integer variable containing the number of lots about to be traded.
OrderID	The ID of the order you are amending, if you are not amending an order leave this field blank.
Price	Price you are amending the order to, if you are not amending the order leave this field blank.
MarginReqd	Address of a string variable containing the ASCII value for the Margin required for the trade about to be made.

The function returns the following error codes:

ptSuccess	Call was successful and data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrUnknownAccount	Supplied TraderAccount name does not refer to a valid record.

3.11.4 ptOpenPositionExposure

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	Exposure	string[21]	writeable, by reference

Returns: Status

The percentage cost of a specific trade when compared to an account's real-time value as presently marked-to market, and can be calculated by the following equation: -

Open Position Exposure (%) = No. of Lots * Margin

100

(Start of Day Net Liquidity Value + P&L)

The *ptOpenPositionExposure* routine is used to retrieve the buying power exposure for a trader account for a given contract. If an invalid contract is passed, then the total exposure for the given trader account is returned. The routine expects the following parameters:

ExchangeName	Address of a string variable containing the ASCII name of the exchange.
ContractName	Address of a string variable containing the ASCII name of the commodity.

ContractDate	Address of a string variable containing the ASCII name of the contract date.
TraderAccount	Address of a string variable containing the ASCII name of the trader account.
Exposure	Address of a string variable containing the ASCII value for the Open Position Exposure, expressed as a percentage of the total available “cash”, SODNLV + P&L.

The function returns the following error codes:

ptSuccess	The call was successful and data has been returned.
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	The API is not logged on to the Host.
ptErrUnknownAccount	The supplied TraderAccount name does not refer to a valid record.

3.11.5 ptPLBurnRate

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	BurnRate	string[21]	writeable, by reference
Returns:	Status		

The *ptPLBurnRate* routine is used to retrieve the buying power burn rate for a trader account for a given Contract.
The routine expects the following parameters:

ExchangeName Address of a string variable for the name of the exchange.

ContractName Address of a string variable for the name of the commodity.

ContractDate Address of a string variable for the name of the date.

TraderAccount Address of a string variable for the name of the account.

BurnRate Address of a string variable to write the Burn Rate to.

The Burn Rate expresses the percentage of the Trader Account’s equity (cash) that is being lost or expended, and can be calculated by the following equation:
BurnRate (%) = (P&L / Start of Day Net Liquidity Value) * 100.

The function returns the following error codes:

ptSuccess	Call was successful
ptErrNotInitialised	API has not been initialised.
ptErrNotLoggedOn	API is not logged on to the Host.
ptErrUnknownAccount	Supplied TraderAccount name does not exist
ptErrUnknownContract	Contract is not recognised

3.11.6 ptGetMarginPerLot

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	MarginReqd	string[21]	writeable, by reference
Returns:	Status		

The *ptMarginPerLot* routine is used to retrieve the margin per lot for a trader account for a given contract. All parameters are mandatory and must be for a valid contract.

3.11.7 ptTotalMarginPaid

Arguments:	ExchangeName	string[11]	read-only, by reference
	ContractName	string[11]	read-only, by reference
	ContractDate	string[51]	read-only, by reference
	TraderAccount	string[21]	read-only, by reference
	MarginReqd	string[21]	writeable, by reference
Returns:	Status		

The *ptTotalMarginPaid* routine is used to retrieve the total margin for a trader account or for a trader account on a given Contract. If no exchange name or contract name or contract date is given, the routine will calculate total margin for the given trader account. However, if the contract details are specified then total margin will be for the specified account and contract.
TraderAccount parameter is mandatory and must be a valid account name. To get total margin for a contract, ExchangeName, ContractName and ContractDate must be specified. If any of the three parameters are blank the routine will calculate total margin for the given trader account.

Appendix A – API change history

Original API version is v1.0

Changes introduced in v1.1

- The following new API routines have been introduced in v1.1
  - ptRegisterContractCallback
  - ptContractAdded (callback)
  - ptContractDeleted (callback)
  - ptSetClientPath
  - ptSetInternetUser
  - ptSetHandshakePeriod
  - ptGetContractByName
  - ptGetContractByExternalID
  - ptGetExtendedContract
  - ptGetExtendedContractByName
  - ptReportTypeExists
  - ptCountOrderHistory
  - ptGetOrderHistory
  - ptSetUserIDFilter
  - ptGetErrorMessage
  - ptSetHostHandshake
  - ptSetPriceHandshake
- It is no longer a requirement to register callback functions for all callbacks. Only *ptHostLinkStateChange* and *ptLogonStatus* callbacks are required.
- The internal order list has been modified to be a list of order lists to allow easier filtering of historical orders. Historical orders can only appear in the main order list if there are order history records with no currently active order record. History records must now be retrieved by order index using ptGetOrderHistory.

Changes introduced in v2.8

- The following new API routines have been introduced in v2.8.0
  - ptDisconnect
  - ptDumpLastError
  - ptForcedLogout (callback)
  - ptGetFillByID
  - ptGetTraderByName
  - ptPriceSnapshot
  - ptPriceStep
  - ptRegisterFillCallback
  - ptRegisterStatusCallback
  - ptSnapdragonEnabled
  - ptStatusChange (callback)
  - ptUnsubscribePrice
  - ptEnabledFunctionality
- The following API routines have changed in v2.8.0.
  - ExchangeName field added to ptCancelBuys, ptCancelSells, ptCommodityExists, ptContractExists,

ptGetAveragePrice, ptGetCommodityByName, ptGetContractPosition, ptGetOpenPosition, ptSubcriberPrice, ptGetPriceForContract
  - *ptGetReportSize:* Now includes space for null terminator.
  - *ptOrder (callback):* Added OldOrderID, uses OrderUpdStruct.
  - *ptRegisterCallback:* Added register for *ptForcedLogout*.
  - *ptRegisterOrderCallback:* Removed ability to use *ptFill* here.
  - ptRegisterPriceCallback: Changed PriceUpdStruct
- The following API routines have been removed in v2.8 as obsolete
  - ptIsTradeable Use ptGetTraderByName.
  - ptOrderTypeExists Use ptGetOrderType.
  - ptOrderTypePriceRequired Use ptGetOrderType.
  - ptOrderTypePrice2Required Use ptGetOrderType.
  - ptGetExpiryDate Use ptGetContractByName.
- The following structures have changed in v2.8
  - **TraderAcctStruct** contains new Tradeable field.
  - **OrderTypeStruct** removed SyntheticType, added NumPricesReqd and NumVolumesReqd.
  - **ContractStruct** changed to allow for variable number of legs. ContractDate is now 50 characters.
  - **ContractUpdStruct** changed ContractDate to 50 characters.
  - **PriceUpdStruct** added ExchangeName, extended ContractDate to 50 characters.
  - **FillUpdStruct** new structure used for *ptFill* callback.
  - **OrderUpdStruct** new structure used for *ptOrder* callback.
  - **FillStruct** added ExchangeName, removed Value.
  - **PriceDetailStruct** added Direction, replaced Timestamp with Hour, Minute and Second fields.
  - **PriceStruct** added Status and Mask to end of structure.
  - **NewOrderStruct** added ExchangeName and OpenOrClose. Removed OCO and Crosstrade. ContractDate is now 50 characters.
  - **AmendOrderStruct** added OpenOrClose. Removed OCO and Crosstrade.
  - **OrderDetailStruct** added ExchangeName and OpenOrClose. Removed OCO and Crosstrade. ContractDate is now 50 characters.

NOTE: The **Patsystems** Order ID provided by the API is a String value. It may contain non-numeric characters, and the numeric section of the Order ID may not be unique (eg. It is possible to have one new Order ‘N1’ and one synthetic called ‘S1’. Synthetic Orders also no longer use a negative Order ID. If your application previously converted the Order ID to an Integer (int) for any reason, that code will no longer be valid.

Changes introduced in v2.8.1

- The following new API routines have been introduced in v2.8.1 to support Cash Margining.
  - ptPLBurnRate
  - ptOpenPositionExposure
  - ptBuingPowerRemaining
  - ptBuyingPowerUsed
  - ptMarginForTrade

The documentation for ptOrder (callback) has been updated to correctly describe the arguments that's should be passed to the API.

Changes introduced in v2.8.2

- The following new API routines have been introduced in v2.8.2
  - ptRegisterExchangeRateCallback
  - ptRegisterConStatusCallback
  - ptRegisterOrderCancelFailureCallback
  - ptSetOrderCancelFailureDelay
- The following structures have changed in version 2.8.2
  - **OrderTypeStruct a**dded NumDatesReqd.
  - **LogonStruct** added Reports.
  - **MessageStruct** fieldv MsgText is now 500 characters.
  - **NewOrderStruct** added GoodTillDate.
  - **OrderDetailStruct** added GoodTillDate.

Changes introduced in v2.8.3

- The following new API routines have been introduced in v2.8.3
  - ptRegisterAtBestCallback
  - ptAtBest (callback)
  - ptRegisterExchangeRateCallback
  - ptCountContractAtBest
  - ptGetContractAtBest
  - ptGetContractAtBestPrices
  - ptSuperTASEnabled
  - ptSetSuperTAS
  - ptSetSSL

The order cancel functions now cancel orders that are closest to market first (by comparing limit price to last traded price).

Changes introduced in v2.8.3-3

The various service pack releases from V2.8.3 to V2.8.3 SP3 contain code enhancements or corrections and do not contain any changes to the interface.

Changes introduced in v2.8.3-4

The SP4 API was released but the ticker is not fully supported and should be considered a work in progress.

- The following new API routines have been introduced in v2.8.3 SP4
  - ptRegisterTickerCallback
  - ptTicker (callback)

Changes introduced in v2.8.3-5

- The following new API routines have been introduced in v2.8.3-5
  - ptRegisterSubscriberDepthBroadcast
  - ptSubscriberDepthUpdate (callback)
  - ptSubscribeBroadcast
  - ptUnsubscribeBroadcast
  - ptCountContractSubscriberDepth
  - ptGetContractSubscriberDepth
  - ptSetMemoryWarning
  - ptMemoryWarning (callback)
  - ptNextOrderSequence
  - ptCreateStrategy
- The following routines are still a work in progress, although the interface is considered defined. Full functioning versions of this functionality may need specific versions of the Market Data Server in order to work:
  - ptRegisterTickerCallback
  - ptTicker (callback)
- The following structures have changed in version 2.8.3-5
  - **OrderTypeStruct** added AutoCreated field
  - **OrderDetailStruct** added Reference field

Changes introduced in v2.8.3-5.2.x

- The following structures have changed in v2.8.3-5.2.x
  - **ExchangeStruct** includes new fields CustomDec and Decimals

Changes introduced in v2.8.3-5.4

- Additional debugging was added for synthetic orders
- The logon message is now encoded
- The following routines have added in v2.8.3-5.4
  - ptSetMDSToken
  - ptSetSSLClientAuthName

Changes introduced in v2.8.3-5.6

- The following routines have been added in v2.8.3-5.6
  - ptSetSSLCertificateName
  - ptSetSSLClientAuthName

Changes introduced in v6

- Order sub-states are now supported along with the Core V2.8.3-6 back end servers.
- New error codes ptErrNoPreallocOrders (40), ptErrDifferentMarkets (41) and ptErrDifferentOrderTypes (42) added.

- New constants ptFillGroup, ptLegGroup and ptOrderGroup added
- New constants for Pre-allocated Order ID status added
- The following structures have changed in v2.8.3-6
  - **ExchangeStruct** includes new fields CustomDec and Decimals
  - **OrderTypesStruct** contains new fields TimeTriggered, RealSynthetic and GTCFlag
  - **LogonStatusStruct** contains new fields DOMEnabled, PostTradeAmend and Username.
  - **NewOrderStruct** contains new fields ESARef, Priority, TriggerDate, TriggerTime, BatchID, BatchType, BatchStatus.
  - **AmendOrderStruct** contains new fields Priority, TriggerDate, TriggerTime, BatchID, BatchType, BatchStatus.
  - **OrderDetailStruct** contains new fields Priority, TriggerDate, TriggerTime, SubState, BatchID, BatchType, BatchStatus
  - **CrossingOrderIDs** is a new data structure
- The following new API routines have been changed or introduced in v2.8.3-6.
  - ptGetAPIBuildVersion
  - ptGetOptionPremium
  - ptGetContractRating
  - ptAddCrossingOrder
  - ptAddOrderEx

Changes introduced in v6.1

- Support has been introduced for variable tick sizes in contracts
- Support has been introduced for block orders
- Support has been introduced for the Order Management Interface (OMI) in the form of aggregate or care orders.
- Support for multi-leg orders up to 16 legs has been improved
- The following new routines have been added in v2.8.3-6.1
  - ptSetSSLCertificateName
  - ptSetSSLClientAuthName
  - ptRegisterVTSCallback
  - ptGetVTSItem
  - ptAddAggregateOrder
  - ptAddBlockOrder
  - ptAddCrossingOrder
  - ptAddBlockOrder
  - ptAddBasisOrder
  - ptAddAAOrder
  - ptGetContractRating (removed)
  - ptVTSUpdate (callback)

- The following structures have changed in v2.8.3-6

- **ExchangeStruct** includes new fields TicketType, RFQA, RFQT, EnableBlock, EnableBasis, EnableAA, EnableCross
- **FillStruct** contains new fields ExchangeFillId, ExchangeRawPrice, ExecutionID to support the Reuters FX project.
- **NewAggOrderStruct** is a new structure to support ptAddAggregateOrder for OMI.
- **NewOrderStruct** contains new fields ParentID, DoneForDay, BigRefField and Reuters FX fields SenderLocationID, RawPrice, RawPrice2, ExecutionID and ClientID.
- **AmendOrderStruct** contains new fields ParentID, DoneForDay, BigRefField, ESARef and Reuters FX fields SenderLocationID, RawPrice, RawPrice2, ExecutionID and ClientID
- **OrderDetailStruct** contains new fields ParentID, DoneForDay, BigRefField, ESARef, QuoteID, Timeout and Reuters FX fields SenderLocationID, RawPrice, RawPrice2, ExecutionID and ClientID
- **VTSDetailStruct** is a new structure to support variable tick sizes.
- **LegPriceStruct** is a new structure to better support multi-leg orders.
- **BasisOrderStruct** is a new structure for supporting block, basis and against actuals, reporting off exchange trades on Connect.

Changes introduced in v6.2

- New order state ptTransferred included in header file
- Introduced large number of additional helper data types for parameter passing.
- Added further support for OMI.
- The following new routines have been removed in v2.8.3-6.1
  - ptSnapdragonEnabled
  - ptVTSUpdate
  - ptGetVTSDetails
- The following new routines have been added in v2.8.3-6.1
  - ptOMIEnabled
  - ptAddCustRequest
  - ptReParent
  - ptDoneForDay
- The following structures have changed in v2.8.3-6
  - **ContractStruct** field Leg renamed to ExternalID for clarity.
  - **NewCustReqStruct** is a new structure to support Customer Requests.
  - **OrderDetailStruct** contains fields ISINCode, CashPrice, Methodology, BasisRef, EntryDate and EntryTime to assist in registering off exchange trades on Connect.

Changes introduced in v6.3

- New additional error messages added:
  - ptErrOrderAlreadyAmending
  - ptErrNotTradableContract
  - ptErrFailedDecompress

- The following new callback types have been added:
  - ptDOMUpdate = 21
  - ptSettlementCallback = 22
  - ptStrategyCreateSuccess= 23
  - ptStrategyCreateFailure= 24
  - ptAmendFailureCallback = 25
  - ptGenericPriceUpdate = 26
  - ptBlankPrice = 27
  - ptOrderSentFailure = 28
  - ptOrderQueuedFailure = 29
  - ptOrderBookReset = 30

- The ContractStruct and ExtendedContractStruct have had the Tick information added to them. The Commodity structure still has this information for legacy reasons.
  - ptOMIEnabled
  - ptAddCustRequest
  - ptReParent
  - ptDoneForDay
- The following structures have changed in v2.8.3-6
  - **PriceStruct** structure has had a number of additional price types added, including indicative and settlement prices.
  - All **OrderStruct structures** have Connect 9.0 relevant fields added to them
  - **StratLegStruct** contains the ContractName to allow InterCommoditySpreads to be identified and added
- The following methods have been added:
  - ptGetGenericPrice
  - ptSetOrderSentFailureDelay
  - ptSetOrderQueuedFailureDelay
  - ptRegisterOrderBookReset
  - ptRegisterStrategyCreateFailure
  - ptRegisterStrategyCreateSuccess
  - ptRegisterSettlementCallback
  - ptRegisterAmendFailureCallback
  - ptRegisterAmendFailureCallback
  - ptRegisterOrderQueuedFailureCallback
  - ptRegisterOrderSentFailureCallback
  - ptRegisterDOMCallback
  - ptRegisterGenericPriceCallback
  - ptRegisterBlankPriceCallback

Functional Description for 6.3 API :Connect 9.0

Enhanced Automated Price Injection Model (APIM)

Permit the assignment of the API and APIM User for the placement of orders and transmit the values to the STAS. APIM flags will be:

- **M** - Manual
- **A** - Automatic

The API will require that an ITM is set for non Patsystems applications when being initialised. For third party applications, this APIM value will always be reverted to 'G' irrespective of the value passed to the Trading API. The application ID of the application placing the order will be validated against Patsystems Application IDs, and if it is not found, the APIM value will be defaulted to 'G'.
The message passed to the STAS for this order will be modified to include the APIM and APIMUser information. These will be appended to the Add Order message (message ID 3301), with the tags '*APIM'* for the APIM value and '*APIMUser*' for the ITM code (passed in the APIM user parameter above).
Additionally, the API will store the APIM and APIMUser information for each Order in the internal OrderQ.

Inter Commodity Spreads – Strategy Creator

The Patsystems internal representation of this will be a contract date with two externals (near leg / far leg) and the strategy code U. The name for the contract date will need to include the ratio between the legs since multiple ICS spreads can exist with differing ratios on the same deliveries.
The strategy creation message being sent to the core will be extended to include the user ID. This will be populated by the API.
When a strategy creation message has been sent to the core, a message will be received by the API describing the status of the created strategy – i.e., whether it has been successfully created, or rejected by the core components/exchange. This will consist of 2 messages – a ClientStrategySuccess and a ClientStrategyFail message.
When these messages are received, the new callbacks StrategyCreateSuccess (23) and StrategyCreateFail (24) will be made to confirm the strategy status (i.e., accepted by the exchange but not available for trading, or rejected by the exchange).

Inter Commodity Spreads – Order Entry

The ICS prices will be NSP as that is the exchange format for ICS prices and will permit SyOMS to be used with ICS orders. ICS prices are quoted in tick increments of 10,000 but with typical denominators of 128. They should be entered and displayed to a resolution of 10,000. This may require the creation of a new TPP / MPM convention for processing purposes. ICS contracts will be risked in the same manner as existing spreads.
ICS prices can be displayed either as the Strategy Net Price (SNP) which is the raw price from the exchange or Net Change Value (NCV).
SNP is officially:
((Front Leg Price – Front Leg YDSP) * Front Leg Ratio) +
((Back Leg Price – Back Leg YDSP) * Back Leg Ratio)

NBB Front Leg Ratio is 1 for all ICS so far defined
NCV is:
SNP – YDSP (for the strategy).
The exchanges recommend that NCV or SNP display be user selectable.
SNP is used for all exchange quotes and order price and consequently will be the PATS internal price format.
ICS orders require leg prices for the underlying contracts these will be carried in the bid price and offer price fields of the PATS order structure.
The convention used will be that the bid price will be the first leg price and the offer price will be the far leg price.
The method ptAddOrder and ptAmendOrder method will be changed to include the front and back leg prices, and the Yesterdays Settlement price for the contract, used to calculate the NCV.

Inter Commodity Spreads – Order Amend

The API will pass through the prices entered for the amended order.

Yesterday's Daily Settlement Price (YDSP)

The API will be revised to accept YDSP and TSP (type 24 and 25), which will be available under revised call-backs.
If the existing settlement price call back is executed the settlement price provided will be TSP or YDSP if TSP is not set.
Change on day will be calculated from YDSP.

Enhanced Depth Of Market

The system is currently restricted to showing DOM up to 10 deep. This is less than the actual depth available form Connect and other exchanges.
There will be a new price depth message that will transmit depths up to 20 deep.
The API will be extended to hold up to 20 bid price combinations and up to 20 offer price combinations.

Functional Description for 6.3 API : CME €$

The Trading API has been enhanced to take advantage of the extended CME Euro Dollar options available for trading. This has had a noticeable impact on the functionality available in the Trading API.

Reduced Tick Contracts

The API uses the tick_size and ticks_per_point for formatting order prices and for calculating the P&L. Currently this is done by referencing the ticks per point for the commodity associated with the contract date, however this will need to be moved to the contract date level.
The Contract Date message received from the STAS (message ID 3658) will have the fields containing Ticks Per Point and TickSize populated. The tag values for these are 'TPP' and 'TSz' respectively. This will only be present if the values for the contract date in question are different to those for the parent commodity.
The Contract Date object will have these two additional properties added to them, and the read methods will return these values if present – otherwise they will return those for the parent Commodity.

These values will be accessed from the Contract Date for all cases where the tick size for a Contract Date is retrieved from the Contract Dates parent Commodity.
The ContractStruct used in methods such as ptGetContract, ptGetontractByName, and ptGetContractByExternalID will be extended to include the TicksPerPoint.

Pseudo Level-2 Depth

The CME have enhanced the bid/offer message (FIX message MA) to include the number of active orders making up the book quote. This is akin to level 2 depth where the counterparty is displayed, but without the actual counter party being explicitly identified. So for example say the bid of 100 lots @ 90 was made up of the following order 50, 20,15,10,5 the message would contain 5 to signify that 5 bids made up the volume. Currently this will have no impact on the client applications

Indicative Prices

The indicative prices are to be sent from the STAS using the Generic Price Message. On receipt the API will update the price structure, and if the structure changes a price callback will be made to the client application.

The Price Structure passed out from the API in methods like ptGetPrice and ptGetPriceForContract will be changed.

Enhanced Quote Request – RFQ Entry

The API will receive two new order types – RFQi and RFQt for the CME exchange.
Unlike existing supported RFQ orders, the CME RFQ's support different sides for an order

Enhanced Quote Request – Price Updates.

The API will receive a new price message from the PDD. This will contain RFQ price information as described by the Exchange, Commodity, Contract Date, Price Type and Side. These will be sent to the client via a new Callback

PDD Synchronise Message

The TMPriceSynchronise message passed from the API to the PDD will need to be modified to tell the PDD to send through the additional Generic price messages. This will pass the value '2' between the EnableToken and UserName fields passed in the message

API Changes

The API will have the additional EnableEurodollar flag added to the ptFunctionalityEnabled method. This will replace the existing EnableSnapdragon flag currently received from the ORE. This will be persisted to the client in the ptEnabledFunctionality method in place of the SnapDragon flag.
If the EnableEurodollar flag is not enabled, the API will compare the strategy code received for an a contract on an exchange with the TradeTicket flag set to 'E'. If the code is excluded, the contract will have a

new ‘Tradable’ field in the Struct set to false. This will be used by the API to ensure new or amended orders received from the client on non-tradable contracts will be rejected.
The contracts will still have to be persisted to the client, as orders on Contracts they cannot trade in will could be received from inside the TAG. Also, any RFQ or indicative price updates received will not be persisted for contracts that are not tradable.
This will require some hardcoding of strategy codes which is contrary to our current design approach, but at this time, it is unavoidable.

Price Blanking

Previously, when an exchange blanks prices, the PDD could be configured to selectively blank fields in the price messages for each tradable instrument. With the number of contracts received for the Eurodollar options, this will be modified to a new Price Blanking message issued by the PDD that will contain the Exchange (compulsory), Commodity (optional) and Contract Date (optional) to be blanked.
If only the Exchange field is populated, all prices for that exchange are to be blanked. Likewise, if the Commodity is also populated, then all Contract Dates for that commodity are to have their prices blanked. Finally, if the Contract Date field is populated, then only that contract date will be blanked.
The new message will be received from the PDD, and the Exchange, Commodity and Contract Date specified by the message will have the prices blanked. If no Commodity is specified, then all Commodities for that Exchange will be blanked, and likewise, if no Contract Date is specified but a Commodity is, all Contract Dates for that Commodity will be blanked. A Price Blank Callback will be made to the client application.

Changes introduced in v7.1

- The following methods have been amended:
  - ptAddCrossingOrder – Additional FAK parameter amended.
- The following methods have been added:
  - ptSubscribeToMarket
  - ptUnsubscribeToMarket

Functional Description for 7.1 API

To provide more efficient electronic trading for equity, foreign exchange and interest rate options, CME have implemented a series of enhancements to their Globex electronic trading platform. A number of changes are required to meet the requirements in the FRS “CMS Globex €$ options requirements”.
The functional changes imposed by the FRS are

- Indicative Requests for Quotes (RFQ’s)
- Indicative Prices
- Options strategies
- Subscription to Active Contracts
- Options navigator for contract selection

Options trading screen(s)
The CME Phase 1 development has limited impact on API development. The two distinct areas of change are:

- Diagnostics written during the downloading of a large number of contracts
- Monitoring of RFQ’s and Last Traded prices for a given Exchange, Contract or Contract Date.

This document details the changes required to the API, and does not include changes required to any other components. However, the interface to the client applications, and messages to the PDD need to be consistent between the respective components.

Diagnostic log file changes

Some additional logging should be performed to assist analysis of the log file when required. During the download, the API writes out all of the information received regarding the contract data.
The main problem with this is the scale of data received generates a significant log file (can be upwards of 60 Mb for 100 k contracts). The logging needs to be reduced in scale to generate a smaller log file, but still retain the scope of information required to allow investigation into issues raised by customers.

Active Market Monitoring

The CME Phase 1 requirements include monitoring RFQ’s and Last Traded Prices for user specified Exchanges, Contracts or Contract Dates. The API will expose two methods to subscribe and unsubscribe to markets on the PDD. When the Subscribe method is called, the API will process the request, and pass it to the PDD. Once done, the API will start receiving price updates from the PDD for Last Traded prices and RFQ’s for the contracts specified. When the client application no longer wants to see this information, the Unsubscribe method will be called, and similarly, the API will pass the equivalent unsubscribe message to the PDD.

Changes introduced in v7.2

- The following methods have been amended:
  - **ptAddOrder**: additional Ghost and Iceberg order parameters included.

- **ptAmendOrder:** additional Ghost and Iceberg order parameters included.
- **ptGetOrder**: structure modified to include Ghost and Iceberg order details

Functional Description for 7.2 API

The API has been modified to support the introduction of Ghost orders and IceBerg orders to the SyOMS functionality. The API now allows the user to place these orders describing the size of clips to be added, and whether consistently sized clips are placed or random clips are placed. The maximum size of the clip to be placed can be amended.

Changes introduced in v7.3

The changes to Trading API were internal, and relevant to changes to the PatSystems trading GUI's. There is no impact to 3rd party users.

Changes introduced in v7.4

- The following method has been added:
  - **ptAddProtectionOrder:** additional method allowing the client application to place protected order types
- The following methods have been amended:
  - **ptAddOrder:** additional protection order parameters included.
  - **ptAmendOrder:** additional protection order parameters included.

Functional Description for 7.4 API

The Trading API has been modified to support the bracket functionality introduced as part of 2.14 SyOMS. The addition of the ptAddProtectedOrder method allows the user to place orders of the type 'Protect!'. SyOMS will place orders to protect the position taken as part of this protected order, and these will have a parent ID of the dummy order held by SyOMS. The order hierarchy is:

- SyOMS generated dummy order used as a parent to all placed order types.
  - Order placed by the user. Identified by the Order type 'Protect!' and has a parent ID of the order placed by SyOMS
  - Level 1 protection order (if details are passed by the client application when ptAddProtectedOrder is called). Identified by being a limit order, and having a parent order ID of the dummy SyOMS order
  - Level 2 protection order (if details are passed by the client application when ptAddProtectedOrder is called). Identified by being a limit order, and having a parent order ID of the dummy SyOMS order
  - Level 3 protection order (if details are passed by the client application when ptAddProtectedOrder is called). Identified by being a limit order, and having a parent order ID of the dummy SyOMS order

Stop protection order. Identified by being a **Stop**! or **TrailingStop!** order type, and having a parent order ID of the dummy SyOMS order.

Changes Introduced in V7.6

- OFSequence number added to the Order update callback and method PtGetOrderByID and the Order callback

Changes Introduced in V7.8

- Passwords can now be encrypted using SHA256 alogorithm
- Cross link is now supported by the API

Changes Introduced in V7.9.6

- 3.3.16 ptGetContract :
  - Added Margin [20] to Margin per lot fields
- 3.5.7 ptAddOrder: added
  - Exchange Field
  - BOFID
  - Badge
  - LocalUserName
  - LocalTrader
  - LocalBOE
  - LocalOrder ID
  - LocalExAcct
  - RoutingID1
  - RoutingID2

Changes Introduced in V8.2.1

- **New callbacks added:**
  - Exchange Update Callback
  - Commodity Update Callback
  - ContractDate Update Callback
  - Purge Completed callback
  - Trader Account Added Callback
  - Ordere Type Update Callback

- **New Order State**
  - ptExternalCancelled

- **New Fill Sub Types**
  - Settlement Fill
  - Minute Fill
  - Underlying Fill
  - Reverse Fill

- Settlement Price Types
  - Exchange For Physical Volume - ptEFPVolume
  - Exchange For Swap Volume - ptEFSVolume
  - Block Volume - ptBlockVolume
  - Exchange for Physical Cummulative Volume - ptEFPCummVolume
  - Exchange For Swap Cummulative Volume - ptEFSCummVolume

  - Block Cummulative Volume - ptBlockCummVolume

- **Additional state – Global Trading States**
  - ptGTUndefined = 0;
  - ptGTActive = 1;
  - ptGTInactive = 2;
  - ptGTExpired = 3;

Changes to structs – some of these changes are extensive so please check the structures in the documentation above:

  - Exchange Struct
  - Commodity Strutc
  - Contract Struct
  - Extended Contract Struct
  - Logon Status Struct
  - Settlement Price Struct
  - OrderDetail Struct

- **Additional structures**
  - Exchange Update Struct
  - Commodity Update Struct

- **New Methods:**
  - ptRegisterExchangeCallback
  - ptRegisterCommodityCallback
  - ptRegisterTraderAddedCallback
  - ptRegisterOrderTypeUpdateCallback
  - ptPurge
  - ptgetConsolidatedPosition
  - ptSetPDDSSL
  - ptSetPDDSSLCertificateName
  - ptSetPDDSSLClientAuthName

Changes Introduced in V8.4.4

The changes to Trading API were internal, and relevant to changes to the PatSystems trading GUI's. There is little impact to 3rd party users.

- Changes to structs:
  - Fill Struct
  - Price Strutc

- New Methods:
  - ptLockUpdates
  - ptUnlockUpdates

- The following methods have been disabled:
  - ptPriceSnapshot

Changes Introduced in V8.4.5.2

- Changes to structs:
  - Amend Order Struct
  - Order Type Struct
- The following method has been changed:
  - ptGetOrderType

Updates Introduced in V8.4.5.2

- Changes to structs:
  - Atbestprice Structure

Updates Introduced in V8.4.6

The following functions have received changes (description and/or their structure updated to align with API and patsIntf.h structure definition):

- ptGetAPIBuildVersion
- ptRegisterExchangeCallback
- ptRegisterGenericPriceCallback
- ptRegisterOrderAmendFailureCallback
- ptRegisterOrderCallback
- ptRegisterOrderQueuedFailureCallback
- ptRegisterOrderSentFailureCallback
- ptRegisterOrderCancelFailureCallback
- ptRegisterOrderTypeCallback
- ptRegisterStrategyCreateFailure
- ptRegisterStrategyCreateSuccess
- ptRegisterTickerCallback
- ptRegisterTraderAddedCallback
- ptSubscribeBroadcast
- ptCreateStrategy
- ptGetExtendedContract
- ptGetOptionPremium
- ptGetTrader
- ptOrderTypeUpdate
- ptTraderAdded
- ptLogOn
- ptAddAggregateOrder
- ptAddAAOrder
- ptAddBasisOrder
- ptAddBlockOrder
- ptAddCrossingOrder
- ptAddCustRequest
- ptAddOrder
- ptAddProtectionOrder
- ptAmendOrder
- ptGetContractAtBestPrices
- ptGetGenericPrice
- ptGetOrder
- ptGetOrderByID
- ptGetOrderHistory
- ptOrder (callback)
- ptCountUsrMsgs has his description added

Updates Introduced in V8.4.7

- ptPriceSnapshot has been put back in
- Fixes regarding Margin Rating not being processed
- Borker Intervation (FIX Trading Gateway only)
- Socket Reconnection – If API doesn't receive any data after "x" period of time it will automatically reconnect (period is 2x the time set for the handshake period)
- Fix for Amend Order Type that was not working accordingly

Updates Introduced in V8.7

- pvYDStl added on Settlement callback
- OrderTypeStruct, NewOrderStruct, AmendOrderStruct, OrderDetailStruct changes
- ptErrInvalidAmendOrderType added on Amend order
- Introduction of Algo Server functions, Active/Inactive Orders
- New functions: ptAddAlgoOrder, ptGetOrderEx, ptGetOrderByIDEx, ptGetOrderHistoryEx, ptAmendAlgoOrder, ptActivateOrder, ptDeactivateOrder

Index of Functions

ptAcknowledgeUsrMsg, 104
ptActivateOrder, 130
ptAddAAOrder, 115, 116, 124
ptAddAggregateOrder, 113
ptAddAlgoOrder, 122
ptAddBasisOrder, 116
ptAddBlockOrder, 116
ptAddCustRequest, 117
ptAddOrder, 118
ptAmendAlgoOrder, 127
ptAmendOrder, 125
ptAtBest (callback), 128
ptBlankPrices, 128
ptBuyingPowerRemaining, 169
ptBuyingPowerUsed, 169
ptCancelAll, 128
ptCancelBuys, 129
ptCancelOrder, 130
ptCancelSells, 132
ptCommodityExists, 74
ptContractAdded (callback), 75
ptContractDeleted (callback), 75
ptContractExists, 76
ptCountCommodities, 77
ptCountContractAtBest, 134, 135
ptCountContracts, 77
ptCountExchanges, 78
ptCountFills, 133
ptCountOrderHistory, 133
ptCountOrders, 134
ptCountOrderTypes, 78
ptCountReportTypes, 79
ptCountTraders, 79
ptCountUserMsgs, 104
ptCreateStrategy, 80
ptDataDLComplete (callback), 83
ptDeactivateOrder, 131
ptDisable, 27
ptDisconnect, 27
ptDOMEnabled, 105
ptDoneForDay, 135
ptDumpLastError, 27
ptEnabledFunctionality, 105
ptExchangeExists, 83
ptFill (callback), 136
ptForcedLogout (callback), 28
ptGetAPIBuildVersion, 29
ptGetAveragePrice, 136
ptGetCommodity, 84
ptGetCommodityByName, 86
ptGetContract, 86
ptGetContractAtBest, 137
ptGetContractAtBestPrices, 138
ptGetContractByExternalID, 88
ptGetContractByName, 88
ptGetContractPosition, 139
ptGetContractSubscriberDepth, 140
ptGetErrorMessage, 30
ptGetExchange, 89
ptGetExchangeByName, 90
ptGetExchangeRate, 91
ptGetExtendedContract, 91
ptGetExtendedContractByName, 93
ptGetFill, 142
ptGetFillByID, 143
ptGetLogonStatus, 107
ptGetOpenPosition, 145
ptGetOptionPremium, 94
ptGetOrder, 146
ptGetOrderByID, 152
ptGetOrderByIDEx, 153
ptGetOrderEx, 152
ptGetOrderHistory, 153
ptGetOrderHistoryEx, 154
ptGetOrderType, 95
ptGetPrice, 155
ptGetPriceForContract, 158
ptGetReport, 97
ptGetReportSize, 98
ptGetReportType, 99

ptGetTotalPosition, 159
ptGetTrader, 100
ptGetTraderByName, 101
ptGetUsrMsg, 108
ptGetUsrMsgByID, 109
ptHostLinkStateChange (callback), 30
ptInitialise, 31
ptLogOff, 110
ptLogOn, 110
ptLogonStatus (callback), 111
ptLogString, 32
ptMarginForTrade, 171
ptMemoryWarning (callback), 32
ptMessage (callback), 112
ptNextOrderSequence, 101
ptNotifyAllMessages, 32
ptOMIEnabled, 112
ptOpenPositionExposure, 172
ptOrder (callback), 159
ptOrderChecked, 160
ptPLBurnRate, 173, 174
ptPriceLinkStateChange (callback), 32
ptPriceSnapshot, 161
ptPriceStep, 161
ptPriceUpdate (callback), 162
ptQueryOrderStatus, 163
ptReady, 33
ptRegisterAtBestCallback, 34
ptRegisterBlankPriceCallback, 35
ptRegisterCallback, 35
ptRegisterConStatusCallback, 37
ptRegisterContractCallback, 39
ptRegisterDOMCallback, 40
ptRegisterExchangeRateCallback, 42
ptRegisterFillCallback, 43
ptRegisterGenericPriceCallback, 43
ptRegisterLinkStateCallback, 45
ptRegisterMsgCallback, 45
ptRegisterOrderAmendFailureCallback, 46
ptRegisterOrderCallback, 47
ptRegisterOrderCancelFailureCallback, 49
ptRegisterOrderQueuedFailureCallback, 48
ptRegisterOrderSentFailureCallback, 49
ptRegisterPriceCallback, 52
ptRegisterSettlementCallback, 52
ptRegisterStatusCallback, 54
ptRegisterStrategyCreateFailure, 55
ptRegisterStrategyCreateSuccess, 56
ptRegisterSubcriberDepthCallback, 53
ptRegisterTickerCallback, 57
ptReParent, 164
ptReportTypeExists, 102
ptSetClientPath, 59
ptSetEncryptionCode, 59
ptSetHandshakePeriod, 60
ptSetHostAddress, 60
ptSetHostHandshake, 61
ptSetHostReconnect, 61
ptSetInternetUser, 61
ptSetMemoryWarning, 62
ptSetOrderCancelFailureDelay, 62
ptSetOrderQueuedFailureDelay, 62
ptSetOrderSentFailureDelay, 63
ptSetPriceAddress, 65
ptSetPriceAgeCounter, 65
ptSetPriceHandshake, 66
ptSetPriceReconnect, 66
ptSetSSL, 63, 67
ptSetSSLCertificateName, 63, 67
ptSetSSLClientAuthName, 64, 67
ptSetSuperTAS, 68
ptSetUserIDFilter, 165
ptSnapdragonEnabled, 165
ptStatusChange (callback), 165
ptSubscribeBroadcast, 69
ptSubscribePrice, 69, 70
ptSubscriberDepthUpdate (callback), 166
ptSuperTASEnabled, 70
ptTicker (callback), 166
ptTraderExists, 103
ptUnsubcribeBroadcast, 71
ptUnsubscribePrice, 71
tenable, 28

Document Control

Master Location: *SharePoint*

Document Name: *Client Trading API Application Developer's Kit*

Document Owner: API Product

Controlled Distribution: Yes

Change History / Version Summary

API	Doc. V	Date	Author	Description	Sections affected
V6.3	V1.3	17-Feb-06		Updated for 6.3 release by Owen Gardner	
V6.3	V1.4D1	28-Feb-06		Reviewed and updated for completeness	
V7.1	V1.4D2 V1.4D3	14 Jul 2006 17 Jul 2006		Reviwed and updated for API 7.1 by Owen Gardner. Format updates.	
V7.4	V1.4D4	02-Oct-06		Updated for API V7.4.	
V7.4	V1.5	03-Oct-06		Process guide for live release and increment guide version to 1.5.	
V7.8	V1.6	15-Jan-07		Updated for V7.8 release by Andrew Cloud.	
V7.9.4	V1.7	13-Mar-07		Updated ref previous 7.6 API release by Andrew Cloud. No 7.9.4 changes to document.	
V7.9.6	V1.8	16-Apr-07		Changed email contact addresses (section 1.7). No other edits for this release.	
V7.9.6	V1.9	20-Jun-07		3.3.16 ptGetContract : Added Margin [20] to Margin per lot fields 3.5.7 ptAddOrder: added Exchange Field, BOFID, Badge, LocalUserName, LocalTrader, LocalBOE, LocalOrder ID, LocalExAcct, RoutingID1 and RoutingID2. Updates for API 7.9.6 summarised	
V8.2.1	2.0	25-Sep-07		Refer to Changes Introduced in V8.2.1 section of this guide.	
V8.2.1	2.1	14-Dec-07		Edited description of ptAddOrder, Client ID to: For CME Globex must be populated using unique SenderSubID tag 50 value.	
V8.4.4	2.2	01-May-08		3.4.8 ptLockUpdates 3.4.15 ptUnlockUpdates 3.5.41 ptPriceSnapshot Price and Fill structure updated	
V8.4.5.2	2.3	16-Jul-08		ptGetOrderType parameters changed OrderTypeStruct and AmendOrderStruct changed	
V8.4.5.2	2.4	12-Aug-08		AtbestPrice structure updated	
V8.4.6	2.5	23-Sep-08		Functions description and/or structure changes to align with API and patsintf.h file No cached data	
V8.4.7	2.6	27-Oct-08		ptPriceSnapshot has been put back in Margin rate recalculation	

API	Doc. V	Date	Author	Description	Sections affected
				Broker Intervantion (FIX Trading Gateway only) Socket reconnection Amend order type fix	
V8.7	2.7	01-Dec-08		pvYDStl added on Settlement callback ptErrInvalidAmendOrderType added on Amend order changes on Structures and new functions added (please refer to section “Updates Introducted in V8.7”)	
V8.7.5.2	2.9D1	16-Jun-09	Colin Pringle	Change to structure (ptGetFill)	
V8.7.5.2	2.9D2	16-Jun-09	J. Maynard-Smith	Proofreading 2.9D1 changes New Patsystems logo, and removing logo from footer Updating this table to have Author and Sections Affected columns.	All
V8.7.5	2.9D3	16-Jun-09	Lily Teo	Correct API Version number	Title page & change history
V8.8	2.10D1	03-Jul-09	Colin Pringle	Added new sections	3.6.6 and 3.6.7
V8.8	2.10D2	03-Jul-09	J. Maynard-Smith	Proofreading 2.10D1 additions.	3.6.6 and 3.6.7
V8.8	2.10D3	10-Jul-09	Colin Pringle	Added, to the description of ptGetMarginPerLot: “All parameters are mandatory and must be for a valid contract.”	3.11.6
V8.8	2.10D2	03-Jul-09	J. Maynard-Smith	Proofreading 2.10D3 additions.	3.11.6
V8.8	2.10	29-Jul-09	Pete Burley	Moved to Issue version	All
V8.8	2.11	11-Aug-09	J. Maynard-Smith	lnContractDate: String[10] changed to String[50]	3.3.19, on page 82
V8.8	2.12	30-Sep-09	J. Maynard-Smith	ptSetPDDSSL, ptSetPDDSSLCertificateName and ptSetPDDSSLClientAuthName: Added the sentence, ‘This function is deprecated as PDD server does not support SSL.’ Changed ‘host transaction server’ to ‘Price server’.	3.2.55, 3.2.56 and 3.2.57
V8.8	2.13	16-Apr-10	S.Donovan	Added one more row to table. Renamed document without date.	3.5.43
V8.8	2.14	05-Aug-10	S.Donovan	Changed contact email addresses and removed reference to MSN Instant Messenger ID.	1.7
V8.8	2.15	25-Mar-11	S.Donovan	Changed SenderLocation and APIM descriptions, on Alok’s instructions.	3.5.7, 3.5.11, 3.5.37
V8.8	2.16	06-Jun-12	S. Donovan	Confidentiality section added.	All
V8.8	2.15.1	06-Jun-12	S.Donovan	Renamed V2.15.1 to avoid conflict with draft version.	All
V8.8	2.15.2D1	15-Nov-12	S.Donovan	Incorporated Eldar’s changes to PriceStruct table.	3.10
V8.8	2.15.2D2	19-Nov-12	S. Donovan	Updated string numbers to match Delphi code.	All
V8.8	2.15.2D3	29-Nov-12	S. Donovan	Clarified ptRegisterTickerCallback instruction.	3.2.42
V8.8	2.15.2D4	21-Jan-13	S. Donovan	Deleted ptAmendPending from being mentioned as an order type which may be cancelled. Deleted a duplicated section.	3.2.2 3.4.7

API	Doc. V	Date	Author	Description	Sections affected
				Corrected a typo in ptCountUsrMsgs function.	
V8.8	2.15.2D5	24-Jan-13	S. Donovan	Changed reference to “an order cancel failure callback” to “an order sent failure callback”.	3.2.54
V8.8	2.15.2	07-May-13	S. Donovan	Changed ExecutionID description from String[21] to String[71]. Deleted multiple references to ICSFarLegPrice parameter. Changed EntryTime parameter from String[9] to String[7]. Undrafted.	3.5.37
V8.8	2.15.3D1	20-Dec-13	F.McKenna	Added (Legacy – no longer in use) under pvCurrStl, pvSODStl, and pvNewStl. Changed One point to one point in 2.15.3.	3.10/2.15.3
V8.8	2.15.3D2	24-Dec-13	V. Punjani	Change histroy row 2.15.3D1 to read “Changed One point to one point in 3.3.17 and not 2.15.3. Changed one point to OnePoint.	Change history section 2.15.3D1 3.3.17
V8.8	2.15.3	24-Dec-13	V. Punjani	Undrafted for release.	All
8.8	2.15.4	21-May-14	F.McKenna	Undrafted for release.	All
8.8	2.15.5D1	14-Oct-14	E. Isayev	Updated system requirements information (Windows XP and Windsock library mention removed, Windows 7 and 8 added)	1.1 Introduction
8.8	2.15.5	16-Oct-14	F.McKenna	Undrafted for release.	All

Document Approval / Review

Version	Date	Name and position of proofreader	Purpose of proofreading (information, comment, signoff, etc)	ɔmpl. Y/N
V1.3	28/02/06	Jezza Sutton (Product Mgr)	Product Approval	
1.4D2	14/06/06	Jezza Sutton	Comment	
V2.8	09/01/09	J. Maynard-Smith	Proofreading and formatting	
V2.9	17/06/09	J. Maynard-Smith	Undrafted, following 2.9D3 changes	