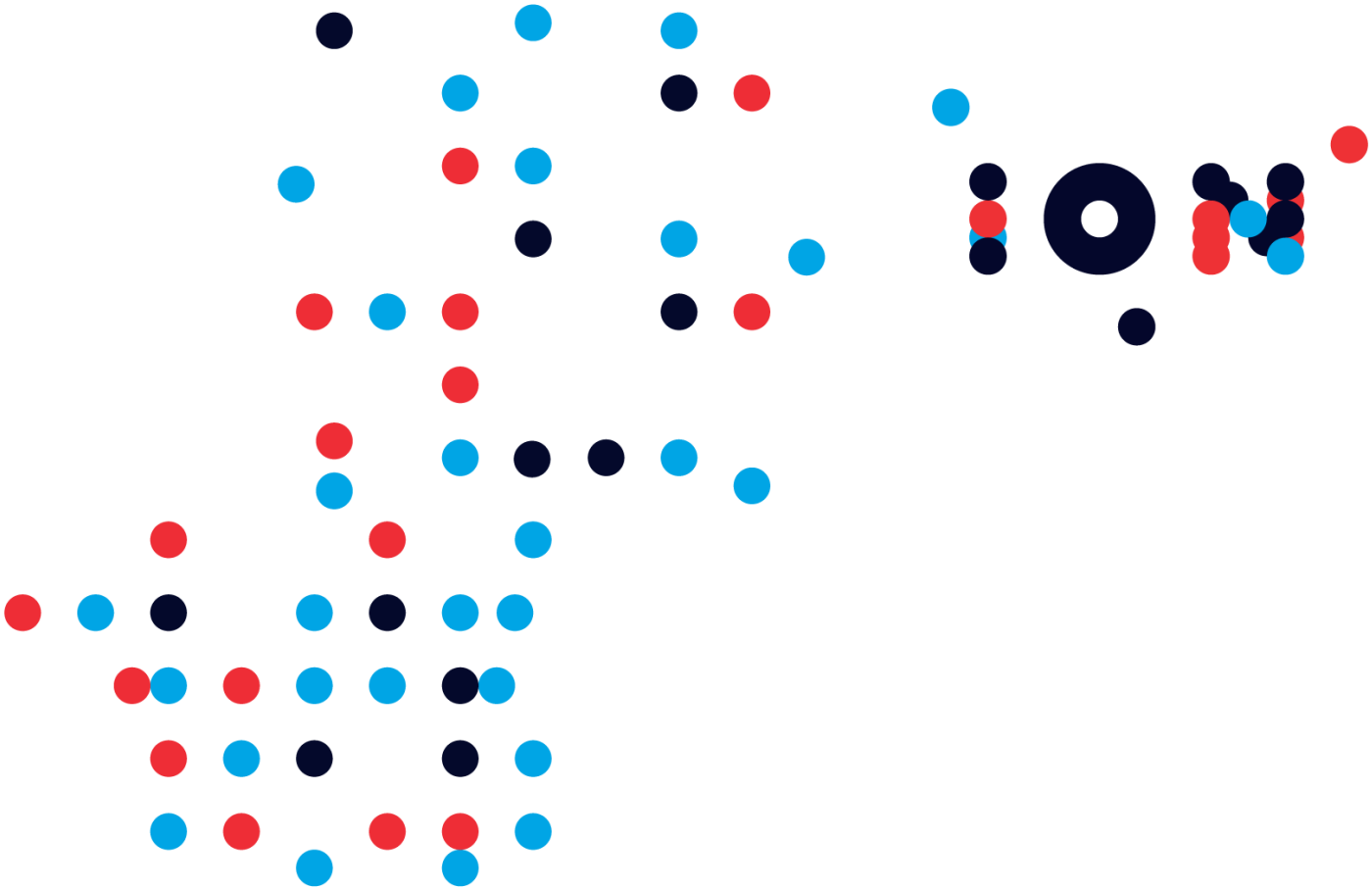
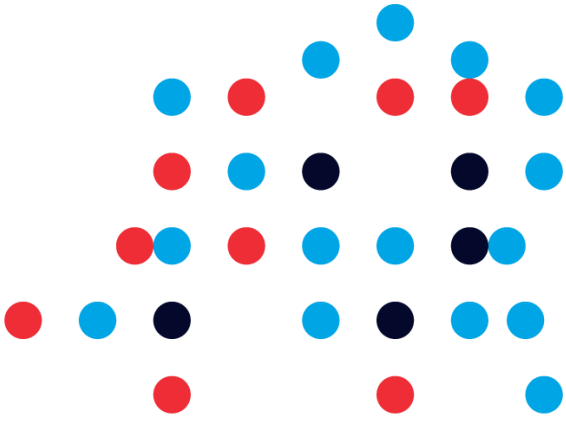


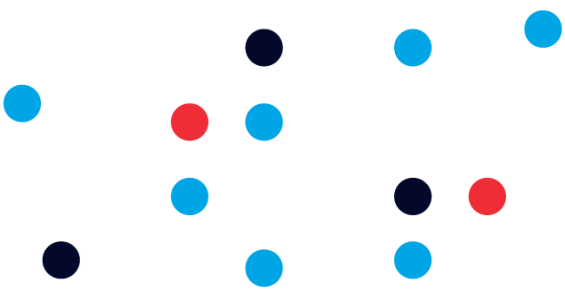
Client Trading API – Application Developer's Kit

API Version 8.9.0+



Developer's guide

Version 1.1



January 2018

Legal Notices

Contents

About this guide

This guide describes how to develop third party applications with the ION:

- Patsystems Client Trading API version 8.9.0+

Audience

This guide is for:

- Third party developers who wish to use the Patsystems Client Trading API to produce their own front end or interface to a Patsystems trading environment.

This guide assumes that you are familiar with Patsystems' trading platform terminology.

Related documentation

- N/A

Conventions

This guide uses several conventions for special terms and actions, operating system-dependent commands, and paths.

Table 1: Conventions

This text formatting …	Is used to indicate …
Bold	Graphical user interface elements.
Italic	• New terms. • Words and phrases that are emphasized. • Cross-references. • Names of documents. • Names of keys.
`Monospace`	• Commands, command options, and flags that appear on a separate line. • Paths and file names. • Code examples and output, and message text. • Variables. • Values you must provide and text strings you must type.

Overview

Introduction

The API has been developed to allow third party applications to interface to the Professional Automated Trading Systems (**Patsystems**) trading engine. Provision to the API is made via 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 7 (32 or 64 bit)
	Windows 8
	Windows 10

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

The API will notify 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 will execute 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 will need to call a query function to obtain the data. For example, the "order updated" callback will return the order-ID. Further details of the order must be obtained from the API by calling the "query order" function.

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.

MiFiD II Support

The API has been enhanced to support the Markets in Financial Instruments Directive (MiFiD) II regulations. An optional parameter has been added to selective functions to allow the explicit setting of the new fields. If the parameter is omitted, then system configured default values will be populated on the messages. The following functions have been changed:

Order Action	Functions
Activate	ptActivateOrder

Amend	ptAmendOrderEx
Cancel	ptCancelOrder ptCancelOrderEx ptCancelOrderEx2 ptCancelBuys ptCancelBuysEx ptCancelSells ptCancelSellsEx ptCancelOrders ptCancelAll ptCancellAllEx
Deactivate	ptDeactivateOrder

To support the regulations the following fields have been introduced:

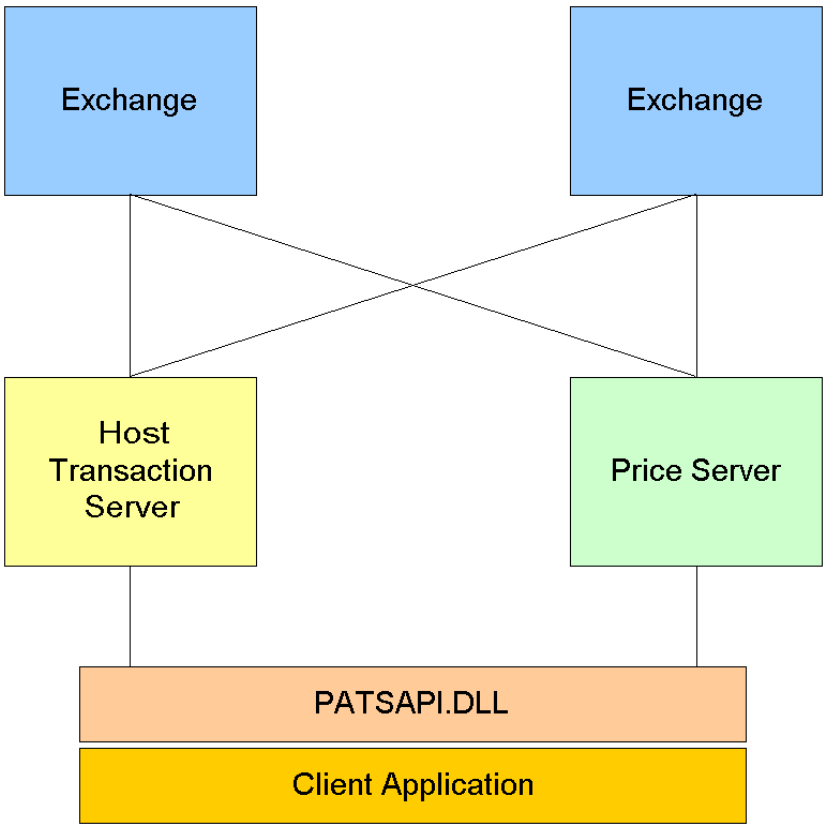
Field Name	Description
clientIdShortCode	Exchange short code
clientIdType	Type of client
commodityDerInd	Commodity Derivative Indicator
DEA	Direct Electronic Access
executionDecision	Exchange short code
executionDecisionType	Type of execution decision
investmentDecision	Exchange short code
investmentDecisionType	Type of investment decision
liquidityProvider	Liquidity provider
shortSelling	Short selling indicator
tradingCapacity	Trading Capacity
ancillaryTrading	Ancillary Trading
fillTimeStamp	Microsecond timestamp YYYYMMDD:hh.mm.ss.ssssss
requestInTimeStamp;	Microsecond timestamp YYYYMMDD:hh.mm.ss.ssssss
requestOutTimeStamp	Microsecond timestamp YYYYMMDD:hh.mm.ss.ssssss
responseInTimeStamp	Microsecond timestamp YYYYMMDD:hh.mm.ss.ssssss
responseOutTimeStamp	Microsecond timestamp YYYYMMDD:hh.mm.ss.ssssss

These new fields will impact the following data structures:

Structure	Used In
NewOrderStruct	ptAddAAOrder ptAddBasisOrder ptAddBlockOrder ptAddCrossingOrder ptAddOrder ptAddOrderEx ptAddAlgoOrder ptAddProtectionOrder
OrderAmendStruct	ptAmendOrder ptAmendOrderEx ptAmendAlgoOrder
OrderDetailStruct	ptCancelOrder ptCancelOrderEx ptCancelOrderEx2 ptCancelAll ptCancelAllEx ptCancelBuys ptCancelSells ptCancelOrders ptActivateOrder ptDeactivateOrder ptGetOrder ptGetOrderEx ptGetOrderByID ptGetOrderByIDEx ptGetOrderHistory ptGetOrderHistoryEx
FillStruct	ptGetFill ptGetFillByID

Patsystems Architecture

A basic understanding of the architecture and terminology used by the Professional Automated Trading System will be useful 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 (a.k.a. the "STAS") 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 given to the client application.

Prices are received from the exchanges and routed to the Price Server (a.k.a. the PDD (Price Data Distributor) which directs them to the API only if requested to do so.

Parameter Passing

The API will accept and return data in one of three formats:

- Binary number
- Single 1-byte character
- 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. 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.

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

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).

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 behavior 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 behavior 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.

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 identify 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 authorized 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.

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 real life but cannot be expected to match exactly). To purchase server time please contact apisupport@patsystems.com and request access to the servers.

Finally, if you do not receive a suitably timely response, or need to provide us with diagnostic files, you can contact us via email on apisupport@patsystems.com.

Guidelines for Development

This section 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.

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:

- 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.
- Initialise 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.
- Set any diagnostic information flags using *ptEnable*. Use these only when initially developing your application, as the functionality can result in large log files (especially price diagnostics).
- Set SSL and the SSL Certificate Name using *ptSetSSL* and *ptSetSSLCertificateName* if SSL is being used.
- 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:
- *ptSetHostAddress*
- *ptSetPriceAddress*
- Register the required callback routines using *ptRegisterLinkStateCallback* for the host connection and *ptRegisterCallback* to register for *ptLogonStatus* and *ptForcedLogout* callbacks.
- Register any other optional callback routines
- Set any other API control parameters. For example, by calling *ptNotifyAllMsgs*
- 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.

Environments

Test and Live Environments

ptInitialise is for specifying which environment you are connecting to and for controlling certain API behaviour. 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.

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

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:

- Wait for *ptHostLinkStateChange* to show "Connected".
- Wait a few seconds (3-5) while API sets encryption detailed structures.
- Issue logon request by calling *ptLogon*.
- Wait for *ptLogonStatus* callback to fire.
- Call *ptGetLogonStatus* to obtain logon status details.
- 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;
```

```
}
```

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.

Tip: [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;
}
```

Price & Market Depth Updates

The Price Server (PDD) 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: 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.

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.

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 state 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*.

Synthetic Orders

Synthetic orders provide Stop or Stop Limit behaviour 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.

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 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.

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 behaviour. 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.

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.

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.

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:

- Call *ptGetReportSize* to obtain size of buffer needed.
- Allocate a contiguous section of memory of the correct size.
- 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.

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 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 many execution orders sent over the course of the day to the exchange.

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 to continue development using the real PATSAPI.DLL to experience correct business flow responses.

The demonstration DLL does not require a license key or application ID and is included in the developer's kit.

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

API Reference

Data Types and Parameters

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

Table 2: Conventions

Type	Description
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.
"char"	Single-byte ASCII character
"byte"	Single-byte integer
"integer"	Four-byte integer
"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.
"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.

Setup Functions

The following functions initialise 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.

ptDisable

This routine disables the diagnostic option specified in the integer bitmask.

Arguments:	• **Code** (integer read only, immediate value)
Returns:	**none**

Argument/Returns	Value
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

ptDisconnect

This routine disconnects the current Host and Price Feed connections.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	Successfully disconnected connections.
	• *ptErrNotInitialised*	API is not initialised.

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 behaviour. 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.

After calling ptDisconnect 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.

ptDumpLastError

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*

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	Successfully completed error dumps.
	• *ptErrFalse*	API is not initialised.

ptEnable

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.

Arguments:	• **Code** (integer read only, immediate value)
Returns:	**None**

Argument/Returns	Value
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.

ptForcedLogout (callback)

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.

Arguments:	**none**
Returns:	**none**

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.

ptGetAPIBuildVersion

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.

Arguments:	• **APIVersion** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successfully return of data

ptGetConsolidatedPosition

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.

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)

Argument/Returns	Value
ExchangeName	Name of the exchange
ContractName	Contract name
ContractDate	Tradeable contract date name
TraderAccount	Trader account name
PositionType	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
Fill	Consolidated position
Status	• *ptSuccess* Successfully completed error dumps. • *ptErrNotInitialised* API is not initialised • ptErrNotLoggedOn User is not logged in • ptErrNoData Contract date specified cannot be found

ptGetErrorMessage

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

Arguments:	**ErrorNo** (interger read only, immediate value)
Returns:	**ErrorMsg** (pointer)

Argument/Returns	Value
ErrorNo	A status code returned by another API routine
ErrorMsg	Message text associated to the ErrorNo

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.

ptHostLinkStateChange (callback)

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.

Arguments:	**Data** (struct writeable, by reference)
Returns:	**none**

Argument/Returns	Value
Data	Address of a structure of type **LinkStateStruct**. The application routine will receive the link status details in this parameter. LinkStateStruct is defined as: **LinkStateStruct read-only, by reference** OldState: byte NewState byte The link states can be one of: ptLinkOpened - socket created ptLinkConnecting - socket connecting to remote socket ptLinkConnected - socket connected to remote socket ptLinkClosed - socket connection has been closed ptLinkInvalid - unknown or unexpected state

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

ptInitialise

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.

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** (integer)

Argument/Returns	Value
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

Argument/Returns	Value
	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
Status	• *ptSuccess* Successfully completed error dumps. • *ptErrNotInitialised* API failed to create data structures. Do not Use • *ptErrWrongVersion* API is not for expected version.

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.

ptLogString

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

Arguments:	• **DebugStr** (string[251], read-only by reference)
Returns:	**status** (integer)

Argument/Returns	Value
DebugStr	Message string to trace in the log file
Status	• *ptSuccess* Successful • *ptErrNotInitialised* API is not initialised.

ptMemoryWarning (callback)

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.

Arguments:	**None**
Returns:	**None**

Note: The callback must be registered with the *ptRegisterCallback* routine, passing in ID *ptMemoryWarning*

ptNotifyAllMessages

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.

Arguments:	• **Enabled** (char immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
Enabled	A char variable containing either Y or N for enable or disable.
Status	• *ptSuccess* Successful • *ptErrNotInitialised* API is not initialised.

ptPriceLinkStateChange (callback)

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.

Arguments:	**Data** (struct writeable, by reference)
Returns:	**none**

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

Argument/Returns	Value
	LinkStateStruct **read-only, by reference** OldState: byte NewState byte The link states can be one of: ptLinkOpened - socket created ptLinkConnecting - socket connecting to remote socket ptLinkConnected - socket connected to remote socket ptLinkClosed - socket connection has been closed ptLinkInvalid - unknown or unexpected 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.

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

ptPurgeCompleted (callback)

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.

Arguments:	**ExchangeData** **(**struct writeable, by reference)
Returns:	**none**

Argument/Returns	Value
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: **ExchangeUpdStruct** **read-only, by reference** ExchangeName: string[11]

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

ptReady

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.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	API has commenced processing.
	• *ptErrCallbackNotSet*	One of the required callbacks has not been provided to the API.
	• *ptErrNotInitialised*	API has not been initialised with ptInitialise.

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: 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.

ptRegisterAtBestCallback

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.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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.

Argument/Returns	Value
	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]
Status	• *ptSuccess* Successful • *ptErrNotInitialised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The *CallbackID* value was not recognised as a valid contract callback.

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.

ptRegisterBlankPriceCallback

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.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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.

Argument/Returns	Value
	BlankPriceStruct **read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11] *ContractDate*: string[51]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The *CallbackID* value was not recognised as a valid contract callback.

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.

ptRegisterCallback

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.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

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

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrUnknownCallback*	The *CallbackID* value was not recognised as a valid contract callback.

This routine is used to register the following callbacks:

CallBack	Description
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.

ptRegisterCommodityCallback

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.

Arguments:	• **CallackID** (integer	read-only, immediate value)
	• **CBackProc** (address	read-only, immediate value)
Returns:	**status** (integer)	

Argument/Returns	Value
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 **one** parameter, of type **CommodityUpdStruct,** passed by reference.

Argument/Returns	Value
	CommodityUpdStruct **read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11]
Status	○ *ptSuccess* Successful ○ *ptErrNotInitilaised* API is not initialised (ptInitialise)

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

ptRegisterConStatusCallback

The ptRegisterConStatusCallback routine registers the callback routine for notifying of a change in connectivity status.

Arguments:	○ **CallackID** (integer read-only, immediate value) ○ **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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 **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]

Argument/Returns	Value	
	See below for field descriptions	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrUnknownCallback*	The *CallbackID* value was not recognised as a valid link state callback.

ConnectivityStatusUpdStruct

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 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

Field	Description
TimeStamp	Date and time that the status message was reported.
SystemID	Globally unique ID identifying the system in which the status message has occured.

ptRegisterContractCallback

The ptRegisterContractCallback routine registers a contract callback routine to notify addition or deletion of contracts.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Register one of the following callbacks: • ptContractAdded — A new contract has been added. • ptContractDeleted — An existing contract has been deleted.
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]
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrUnknownCallback* — The CallBackID value was not recognised as a valid callback.

ptRegisterDOMCallback

The ptRegisterDOMCallback routine registers a contract callback routine to notify the receipt of a Depth Of Market (DOM) message from the price sever

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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 referred to by the DOM message **DOMUpdStruct read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11] ContractDate: string[51]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterExchangeCallback

The ptRegisterExchangeCallback routine registers a callback routine to notify users of an update to an existing exchange.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptExchangeUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **ExchangeUpdStruct,** passed by reference. This structure contains the exchange name. **ExchangeUpdStruct read-only, by reference** *ExchangeName*: string[11]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

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

ptRegisterExchangeRateCallback

The ptRegisterExchangeRateCallback routine registers the callback routine for notification of a change in exchange rate

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptExchangeRate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **ExchangeRateUpdStruct,** passed by reference. This structure contains the currency.

Argument/Returns	Value
	ExchangeRateUpdStruct read-only, by reference *Currency*: string[11]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterFillCallback

The ptRegisterFillCallback routine registers the callback routine for notification of a fill.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptFill**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **FillUpdStruct,** passed by reference. This structure contains the order and fill identifiers **FillUpdStruct read-only, by reference** *OrderID*: string[11] *FillID*: string[71]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

The callback provides the order ID and the Fill ID for the fill that was received.

ptRegisterGenericPriceCallback

The ptRegisterGenericPriceCallback routine registers the callback routine for notification of receipt of a generic price type.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptGenericPriceUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **GenericPriceStruct,** passed by reference. This structure contains the price details **GenericPriceStruct read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11] *ContractDate:* string[51] *PriceType:* integer *BuyOrSell* char See below for price types
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

Price Types

Type	Value	Description
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

Type	Value	Description
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

Note: The price can be retrieved by calling ptGetGenericPrice.

ptRegisterLinkStateCallback

The ptRegisterLinkStateCallback routine registers the callback routine for notification of a link state change.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Register one of the following callbacks: • ptHostLinkStateChange Host server state change. • ptPriceLinkStateChange Price server state change.

Argument/Returns	Value
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **LinkStateStruct,** passed by reference. This structure contains the currency. **LinkStateStruct read-only, by reference** OldState: byte NewState byte The link states can be one of: ptLinkOpened - socket created ptLinkConnecting - socket connecting to remote socket ptLinkConnected - socket connected to remote socket ptLinkClosed - socket connection has been closed ptLinkInvalid - unknown or unexpected state
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterMsgCallback

ptRegisterMsgCallback registers the callback routine for notification of user messages (alerts).

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **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
Status	• *ptSuccess* Successful

Argument/Returns	Value
	• *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterAmendFailureCallback

The ptRegisterAmendFailureCallback registers the callback routine for notifications of an order amend sending failure within a period of time.

Note: 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*.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrderAmendFailure**
CBackProc	Address of a procedure that the API will execute. The 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
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterOrderCallback

The ptRegisterOrderCallback routine registers the callback routine for notification of an order change.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrder**
CBackProc	Address of a procedure that the API will execute. The 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
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

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.

ptRegisterOrderQueuedFailureCallback

The ptRegisterOrderQueuedFailureCallback registers the callback routine for notification of an order sending failure within a period of time.

Note: 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 *pSetOrderQueuedFailureDelay*.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrderQueuedFailure**
CBackProc	Address of a procedure that the API will execute. The 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
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterOrderSentFailureCallback

The ptRegisterOrderSentFailureCallback registers the callback routine for notification of an order sending failure within a period of time.

Note: 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 *pSetOrderSentFailureDelay*.

Arguments:	• **CallackID** (integer read-only, immediate value)

	• **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrderSentFailure**
CBackProc	Address of a procedure that the API will execute. The 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
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterOrderCancelFailureCallback

The ptRegisterOrderCancelFailureCallbackroutine registers the callback routine for notification of an order cancellation failure within a period of time.

Note: 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 *pSetOrderCancelFailureDelay*.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrderCancelFailure**
CBackProc	Address of a procedure that the API will execute. The 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
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterOrderTypeUpdateCallback

The ptRegisterOrderTypeCallback routine registers the callback routine for notification of an ordertype change.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptOrderTypeUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **OrderTypeStruct,** passed by reference.

Argument/Returns	Value
	OrderTypeStruct read-only, by reference *OrderType*: string[11] *Exchange*: 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] Please refer to ptGetOrderType for description of fields
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterPriceCallback

The ptRegisterPriceCallback routine registers the callback routine for notification of receipt of a price change.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptPriceUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **PriceUpdStruct,** passed by reference.

Argument/Returns	Value
	This structure contains the price details **PriceUpdStruct read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11] *ContractDate:* string[51]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

Note: 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.

ptRegisterSettlementCallback

The ptRegisterSettlementCallback registers the callback routine that will fire whenever a settlement price is received by the API.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptSettlementCallback**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **SettlementPriceStruct,** 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.

Argument/Returns	Value
	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]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

Note: There is **no** need to call ptGetPriceForContract, as the last two parameters describe the Settlement Price type received, and the value.

ptRegisterSubscriberDepthCallback

The ptRegisterSubscriberDepthCallback registers the callback routine that will fire whenever subscriber depth of market data is updated.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptSubscriberDepthUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **SubscriberDepthUpdStruct,** passed by reference. This structure contains the exchange name, contract name and date for the data that changed.

Argument/Returns	Value
	SubscriberDepthUpdStruct read-only, by reference *ExchangeName*: string[11] *ContractName*: string[11] *ContractDate:* string[51]
Status	○ *ptSuccess* Successful ○ *ptErrNotInitilaised* API is not initialised (ptInitialise) ○ *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

Subscriber depth is related to At Best prices, and supplies a firm's volume available at a 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.

Note: 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).

ptRegisterStatusCallback

The ptRegisterStatusCallback routine registers the callback routine for notifying of a change in market status of contract dates.

Arguments:	○ **CallackID (**integer read-only, immediate value) ○ **CBackProc (**address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptStatusUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **StatusUpdStruct,** passed by reference.

Argument/Returns	Value
	This structure contains the exchange name, contract name, date and the new status. **StatusUpdStruct read-only, by reference** *ExchangeName*: string[11] *ContractName*: string[11] *ContractDate:* string[51] *Status:* integer See below for Status codes.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

Note: Not all state changes apply to all markets. Many markets do not report state changes through the **Patsystems** servers at all.

Status Code	Description
ptNormal	Exchange open
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

ptRegisterStrategyCreateFailure

The ptRegisterStrategyCreateFailure routine registers the callback routine for notifying of a failure to create a strategy passed in using ptCreateStrategy.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptStrategyCreateFailure**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **StrategyCreateFailureStruct,** passed by reference. **StrategyCreateFailureStruct read-only, by reference** *Username*: string[11] *Exchange*: string[11] *Contract:* string[11] *ContractDate:* string[51] *Text:* string[61]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterStrategyCreateSuccess

The ptRegisterStrategyCreateSuccess routine registers the callback routine for notifying of a successful creation of a strategy passed in using ptCreateStrategy.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptStrategyCreateSuccess**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **StrategyCreateSuccessStruct,** passed by reference. **StrategyCreateSuccessStruct read-only, by reference** *Username*: string[11] *Exchange*: string[11] *Contract:* string[11] *ReqContractDate:* string[51] *GenContractdate* string[51]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptRegisterTickerCallback

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.

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.

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.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptTickerUpdate**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **TickerUpdStruct,** passed by reference. **TickerUpdStruct** *Exchange*: string[11] *Contract:* string[11] *ContractDate:* string[51] *BidPrice:* string[21] *BidVolume:* integer Offer*Price:* string[21] *OfferVolume:* integer *LastPrice:* string[21] *LastVolume::* integer *Bid:* char *Offer:* char *Last:* char See below for description of fields
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

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.

TickerUpdStruct

Field	Description
ExchangeName	Exchange name.
ContractName	Contract name.
ContractDate	Contract date.

Field	Description
BidPrice	The Bid price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CME/CBOT products.
BidVolume	The Bid volume.
OfferPrice	The Offer price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CME/CBOT products.
OfferVolume	The Offer volume.
LastPrice	The Last price. Converts to a floating point number under the rules of the contract. Please be aware of fractional based pricing on some CME/CBOT products.
LastVolume	The Last volume.
Bid	Y or N, to indicate if this message conatins an update to the bid price or bid volume.
Offer	Y or N, to indicate if this message conatins an update to the offer price or offer volume.
Last	Y or N, to indicate if this message conatins an update to the last price or last volume.

ptRegisterTraderAddedCallback

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.

Arguments:	• **CallackID** (integer read-only, immediate value) • **CBackProc** (address read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
CallbackID	Integer to identify the callback routine being provided. Use **ptTraderAdded**
CBackProc	Address of a procedure that the API will execute. The procedure must accept **one** parameter, of type **TraderAcctStruct,** passed by

Argument/Returns	Value
	reference. **TraderAcctStruct read-only, by reference** *TraderAccount*: string[21] *BackOfficeID*: string[21] *Tradeable:* char *LossLimit:* integer
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownCallback* The CallBackID value was not recognised as a valid callback.

ptSetClientPath

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.

Arguments:	• **Path** (string read-only, by reference)
Returns:	**None**

Argument/Returns	Value
Path	The parameter used to specify the client path should be a null terminated string, and should end with a backslash ("\ ") character.

Warning: 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.

ptSetEncryptionCode

The ptSetEncryptionCode routine requests the API to encrypt messages sent to and from the Host Transaction Server.

Arguments:	• **Ecode** (char read-only, immediate value)
Returns:	**None**

Argument/Returns	Value
Ecode	A single character from 'A' to 'E'

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**.

Warning: 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.

ptSetHostAddress

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

Arguments:	• **IPaddress** (string read-only, by reference) • **IPsocket** (string read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
IPaddress	The address of a null terminated string containing the ASCII representation of the IP address including periods.
IPsocket	The address of a null terminated string containing the ASCII representation of the IP socket.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

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.

ptSetHostHandshake

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

Arguments:	• **Interval** (integer	read-only, immediate value)
	• **TimeOut** (integer	read-only, immediate value)
Returns:	**status** (integer)	

Argument/Returns	Value	
Interval	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.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

ptSetHostReconnect

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.

Arguments:	• **Interval** (integer	read-only, immediate value)
Returns:	**status** (integer)	

Argument/Returns	Value
Interval	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.

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

ptSetInternetUser

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.

Arguments:	• **Enable** (char read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value	
Enable	Y or N to enable/disable.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

ptSetMemoryWarning

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.

Arguments:	• **MemAmount** (integer read-only, immediate value)
Returns:	**none**

Argument/Returns	Value
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.

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.

Warning: 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.

ptSetOrderCancelFailureDelay

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).

Arguments:	• **Delay** (integer read-only, by reference)
Returns:	**none**

Argument/Returns	Value
Delay	An integer value containing the number of seconds to wait before issuing the callback.

Warning: 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.

ptSetOrderQueuedFailureDelay

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).

Arguments:	• **Delay** (integer read-only, by reference)
Returns:	**none**

Argument/Returns	Value
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.

ptSetOrderSentFailureDelay

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).

Arguments:	• **Delay** (integer read-only, by reference)
Returns:	**none**

Argument/Returns	Value
Delay	An integer value containing the number of seconds to wait before issuing the callback.

Warning: 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.

ptSetPDDSSL

This function has been deprecated.

ptSetPDDSSLCertificateName

This function has been deprecated.

ptSetPDDSSLClientAuthName

This function has been deprecated.

ptSetPriceAddress

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 *nnn.nnn.nnn.nnn.* **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.

Arguments:	• **IPaddress** (string read-only, by reference) • **IPsocket** (string read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
IPaddress	The address of a null terminated string containing the ASCII representation of the IP address including periods.
IPsocket	The address of a null terminated string containing the ASCII representation of the IP socket.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

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.

ptSetPriceAgeCounter

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.

Arguments:	• **MaxAge** (integer read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

Note: 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.

ptSetPriceHandshake

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.

Arguments:	• **Interval** (integer read-only, immediate value) • **TimeOut** (integer read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
Interval	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.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

ptSetPriceReconnect

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.

Arguments:	• **Interval** (integer read-only, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

ptSetSSL

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.

Arguments:	• **Enabled** (char read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
Enabled	Y to enable SSL
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

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.

ptSetSSLCertificateName

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.

Arguments:	• **CertName** (string[51] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
CertName	Certificate Name	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

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.

ptSetSSLClientAuthName

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.

Arguments:	• **CertName** (string[51] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
CertName	Certificate Name	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

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.

ptSetSuperTAS

The "SuperTAS" host transaction server must be set as enabled.

Arguments:	• **Enabled** (char read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Enabled	Y to enable SuperTAS	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

ptSetMDSToken

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.

Arguments:	• **MDSToken** (string[11] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
MDSToken	Token ID
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

ptSubscribeBroadcast

The ptSubscribeBroadcast requests the Price Server to supply broadcast messages.

Arguments:	• **ExchangeName** (string[11] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Exchange name to receive broadcast messages for.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownExchange* Unknown exchange name

ptUnsubscribeBroadcast

The ptSubscribeBroadcast requests the Price Server to stop supplying broadcast messages.

Arguments:	• **ExchangeName** (string[11] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Exchange name to stop receiving broadcast messages for.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrUnknownExchange* Unknown exchange name

ptSubscribePrice

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.

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)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name to receive prices for.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrUnknownContract* The Exchange, Contract and Date was not recognised.

It is legitimate to subscribe to prices before 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; it is therefore necessary to subscribe to prices each time the API is invoked.

ptUnSubscribePrice

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 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.

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)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name to receive prices for.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrUnknownContract* The Exchange, Contract and Date was not recognised.

ptSubscribeToMarket

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.

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)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name to receive prices for.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrMDSUnavailable* API is not currently connected to a price feed • *ptErrUnknownExchange* The ExchangeName was not recognised • *ptErrUnknownCommodity* The Contract was not recognised • *ptErrUnknownContract* The Date was not recognised.

ptUnsubscribeToMarket

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.

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)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name to receive prices for.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrMDSUnavailable* API is not currently connected to a price feed • *ptErrUnknownExchange* The ExchangeName was not recognised • *ptErrUnknownCommodity* The Contract was not recognised • *ptErrUnknownContract* The Date was not recognised.

ptSuperTASEnabled

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

Returns:	**status** (integer)

Argument/Returns	Value
Status	• *ptSuccess* API is currently connected to a SuperTAS • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* API is not currently logged on to the host. • *ptErrNotEnabled* API is not currently enabled to connect to a SuperTAS.

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:

- Attempt log on by calling *ptLogon*.
- Received log on result notification via *ptLogonStatus* callback.
- Checked log on status by calling *ptGetLogonStatus*
- Received *ptDataDLComplete* callback.

ptCommodityExists

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.

Arguments:	• **ExchangeName** (string[11] read-only, by reference) • **ContractName** (string[11] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name.
ContractName	Address of a string variable containing the ASCII name of the commodity.
Status	• *ptSuccess* Successful commodity exists

Argument/Returns	Value	
	• *ptErrFalse*	Commodity does not exisit
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	No Commodity data in the API

ptCommodityUpdate (callback)

Arguments:	**CommodityData** (struct	writeable, by reference)
Returns:	**none**	

Argument/Returns	Value
CommodityData	Address of a structure of type **CommodityUpdStruct**. The application routine will receive the commodity updated in this parameter. **CommodityUpdStruct** **read-only, by reference** ExchangeName: string[11] CommodityName: string[11]

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

ptContractAdded (callback)

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

Arguments:	**ContractData** (struct	writeable, by reference)
Returns:	**None**	

Argument/Returns	Value
ContractData	Address of a structure of type **ContractUpdStruct**. The application routine will receive the contract updated in this parameter.

Argument/Returns	Value
	ContractUpdStruct **read-only, by reference** ExchangeName: string[11] CommodityName: string[11] Contractdate string[51]

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

ptContractDeleted (callback)

The ptContractDeleted callback fires whenever a contract is removed post logon. The callback returns information that uniquely identifies the contract that has been removed.

Arguments:	**ContractData** **(**struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
ContractData	Address of a structure of type **ContractUpdStruct**. The application routine will receive the contract updated in this parameter. **ContractUpdStruct** **read-only, by reference** ExchangeName: string[11] CommodityName: string[11] Contractdate string[51]

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

ptContractUpdated (callback)

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.

Arguments:	**ContractData** **(**struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
ContractData	Address of a structure of type **ContractUpdStruct**. The application routine will receive the contract updated in this parameter.

Argument/Returns	Value
	ContractUpdStruct **read-only, by reference** ExchangeName: string[11] CommodityName: string[11] Contractdate string[51]

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

ptContractExists

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.

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)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name.
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.
Status	• *ptSuccess* Successful contract exists • *ptErrFalse* Contract does not exisit • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrNoData* No Commodity data in the API

ptCountCommodities

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

Arguments:	**Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Count	Address of an integer variable in which the API will write the result.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.

ptCountContracts

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 the contracts.

Arguments:	**Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Count	Address of an integer variable in which the API will write the result.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.

ptCountOrderTypes

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.

Arguments:	**Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
Count	Address of an integer variable in which the API will write the result.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host.

ptCountReportTypes

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.

Arguments:	**Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
Count	Address of an integer variable in which the API will write the result.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host.

ptCountTraders

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

Arguments:	**Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
Count	Address of an integer variable in which the API will write the result.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host.

ptExchangeUpdated (callback)

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.

Arguments:	**ExchangeData** (struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
ExchangeData	Address of a structure of type **ExchangeUpdStruct**. The application routine will receive the contract updated in this parameter. **ExchangeUpdStruct** **read-only, by reference** ExchangeName: string[11]

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

ptCreateStrategy

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.

Arguments:	• **StrategyCode** (char read-only, immediate) • **NoOfLegs** (integer read-only, immediate) • **ExchangeName** (string[11] read-only, by reference) • **ContractName** (string[11] read-only, by reference)

	• **Legs** (struct read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
StrategyCode	A character containing the appropriate strategy code for the strategy, as listed below.
NumOfLegs	An integer containing the number of legs in the strategy
ExchangeName	Address of a string variable containing the Exchange name
ContractName	Address of a string variable containing the ASCII name of the commodity.
Legs	Address of a structure, type StrategyLegsStruct, containing the legs making up the strategy. Up to 16 legs can be defined.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrInvalidUnderlying* The underlying legs are not valid.

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

StrategyLegsStruct
Leg0: StratLegStruct
Leg1: StratLegStruct

….

Leg14: StratLegStruct
Leg15: StratLegStruct

StratLegStruct

Field	Description
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.

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

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.

Strategy Codes
• **ptDIV_C_CALENDAR,** • **ptDIV_C_SPREAD,** • **ptDIV_CONVERSION,** • **ptDIV_F_SPREAD,** • **ptDIV_P_CALENDAR,** • **ptDIV_P_SPREAD,** • **ptDIV_STRADDLE,** • **ptDIV_STRANGLE.**

ptDataDLComplete (callback)

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.

Arguments:	**None**
Returns:	**None**

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 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.

Note: The callback must be registered with the *ptRegisterCallback* routine.

ptExchangeExists

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

Arguments:	• **ExchangeName** (string[11] read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string variable containing the Exchange name.
Status	• *ptSuccess* Successful exchange exists • *ptErrFalse* Exchange does not exisit • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrNoData* No Commodity data in the API

ptGetCommodity

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.

Arguments:	• **Index** (integer read-only, immediate value) • **Commodity** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.

Argument/Returns	Value	
Commodity	Address of a data structure of type **CommodityStruct** where the API will write the commodity data. This is a packed structure see below for format.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to he host.
	• *ptErrNoData*	No Commodity data in the API
	• *ptErrInvalidIndex*	Value supplied for the index is out of the range of data.

CommodityStruct

Field	Description
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.

Field	Description
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, including fractional priced contracts.

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 cents

ptGetCommodityByName

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.

Arguments:	• **ExchangeName** (string[11] read-only, by reference) • **CommodityName** (string[11] read-only, by reference) • **Commodity** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to he host. • *ptErrNoData* No Commodity data in the API • *ptErrUnknownCommodity* Commodity name given did not match any known records.

ptGetContract

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.

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

Argument/Returns	Value
Index	An integer specifying which record to return. Specify a value between 0 and *ptCountContracts* – 1.
Contract	Address of a data structure of type **ContractStruct** where the API will write the contract data. This is a packed structure see below for format.
Status	• *ptSuccess* Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to he host.
	• *ptErrNoData*	No Commodity data in the API
	• *ptErrInvalidIndex*	Value supplied for the index is out of the range of data.

ContractStruct

Field	Description
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

Field	Description
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] – contract which this contract links to *
InContractDate	String[51] contract date which this contract links to *
ExternalID	An array of 2 LegStruct structures (defined below) containing the first exchange specific contract specification.

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

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

LegStruct

Position	Description
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.

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].

Note: 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.

ptGetContractByExternalID

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

Arguments:	• **ContractIn** (struct read-only, by reference) • **ContractOut** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErInvalidIndex* The values specified do not refer to a valid contract record.

ptGetContractByName

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

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** (integer)		

Argument/Returns	Value	
ExchaneName	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.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	The API does not currently hold any contract information.
	• *ptErInvalidIndex*	The values specified do not refer to a valid contract record.

User Functions

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

ptAcknowledgeUsrMsg

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".

Arguments:	• **MsgID**	(string[11]	read-only, by reference)

Returns:	**status** (integer)

Argument/Returns	Value	
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.	
Status	• *ptSuccess*	Successful
	• *ptErrUnknownMsgID*	Specified message ID does not refer to a valid message.
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	No Commodity data in the API

ptCountUsrMsg

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.

Arguments:	• **Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Count	Address of an integer variable where the API will write the return value.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.

ptDOMEnabled

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.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrUnexpected*	An unexpected error occurred.

ptEnabledFunctionality

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.

Arguments:	• **FunctionalityEnabled** (integer writeable, by reference) • **SoftwareEnabled** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
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 functionalities are enabled for different users.	
Status	• *ptSuccess*	Successful
	• *ptErrUnexpected*	An unexpected error occurred.

FunctionalityEnabled		SoftwareEnabled	
Bit	Meaning	Bit	Meaning
0	DOM Enabled	0	User defined bit

FunctionalityEnabled		SoftwareEnabled	
1	Post Trade Amend Enabled	1	User defined bit
2	MEL Enabled	2	User defined bit
3	Not In Use	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=23, 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)
Value =  23
```

To use the user defined region [0..3] of SoftwareEnabled, you will need to 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 apisupport@patsystems.com for more information.

ptGetLogonStatus

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.

Arguments:	○ **LogonStatus** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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 (see beow).
Status	○ *ptSuccess* Successful

Argument/Returns	Value
	• *ptErrNotInitilaised* API is not initialised (ptInitialise)

LogonStatusStruct

Field	Description
Status	A byte integer indicating the success or failure of the logon. See below.
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 Status field contains one of:

Status Value	Description
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

Status Value	Description
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.

ptGetUserMsg

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.

Arguments:	• **Index** (integer read-only, immediate value) • **UserMsg** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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. (See below).
Status	• *ptSuccess* Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *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.

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*.

MessageStruct

Field	Description
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

ptGetUserMsgByID

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.

Arguments:	• **MsgID**	(string[10]	read-only, immediate value)
	• **UserMsg**	(struct	writeable, by reference)
Returns:	**status** (integer)		

Argument/Returns	Value
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. (See *ptGetUsrMsg*).
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *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.

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*.

ptLockUpdates

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.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise)

ptUnLockUpdates

ptUnlockUpdates unloads the queued list of updates and sends the updates to the client.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

ptLogOff

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*.

Arguments:	**None**
Returns:	**status** (integer)

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptNotLoggedOn*	API has not logged on to the host.

ptLogon

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.

Arguments:	• **LogonDetail**	(struct	read-only, by reference)
Returns:	**status** (integer)		

Argument/Returns	Value	
LogonDetail	Address of a structure of type **LogonStruct** containing the username and password for logging on to PATS. See below for LogonStruct.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErBadPassword*	The value for *NewPassword* is not valid.

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.

LogonStruct

Field	Description
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	Not Used

Field	Description
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). The One Time Password is required if this feature is enabled in the Patsystems Core Server.

ptLogonStatus (callback)

The ptLogonStatus callback executes when the Host has processed the log on request sent using *ptLogon*.

Arguments:	**None**
Returns:	**None**

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.

Note: The callback must be registered using the *ptRegisterCallback* routine.

ptMessage (callback)

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.

Arguments:	○ **MsgID** (string[11] writeable, by reference)
Returns:	**None**

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

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.

Note: The callback must be registered using the *ptRegisterMsgCallback* routine.

ptPostTradeAmendEnabled

This routine is superseded by the *ptEnabledFunctionality* routine.

Trading Functions

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

ptAddAAOrder

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.

Arguments:	• **PrimaryOrder** (struct read-only, by reference) • **SecondaryOrder** (struct read-only, by reference) • **BidUser** (string[11] read-only, by reference) • **OfferUser** (string[11] read-only, by reference) • **OrderIDs** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
PrimaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description.
SecondaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description
BidUser	A String[11] containing the bid user

Argument/Returns	Value
OfferUser	A String[11] containing the offer user
OrderIDs	Address of a structure of type **CrossingOrderIDs.** See below.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrPriceRequired* Order type required a price and one was not provided. • *ptErrUnknownAccount* Trader account does not match known trader. • *ptErrUnknownOrderType* Order type is not known. • *ptErrUnknownContract* Contract name/ date does not refer to a valid contract. • *ptErrTASUnavailable* Transaction server is not connected. • *ptErrMDSUnavailable* PDD is not connected.

CrossingOrderIDs

Field	Description
PrimaryOrderID	string[11] – order id returned for primary order
SecondaryOrderID	string[11] – order id returned for secondary order

ptAddBasisOrder

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

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** (integer)	

Argument/Returns	Value	
PrimaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description.	
SecondaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description	
BasisOrder	Address of a structure of type **BasisOrderStruct**, see below.	
OrderIDs	Address of a structure of type **CrossingOrderIDs.** See below.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrPriceRequired*	Order type required a price and one was not provided.
	• *ptErrUnknownAccount*	Trader account does not match known trader.
	• *ptErrUnknownOrderType*	Order type is not known.
	• *ptErrUnknownContract*	Contract name/ date does not refer to a valid contract.
	• *ptErrTASUnavailable*	Transaction server is not connected.
	• *ptErrMDSUnavailable*	PDD is not connected.

BasisOrderStruct

Field	Description
ISINCode	string[21] – ISIN code
CashPrice	string[21] – cash price
Methodology	char - methodology
Reference	String[21] – reference text

CrossingOrderIDs

Field	Description
PrimaryOrderID	string[11] – order id returned for primary order
SecondaryOrderID	string[11] – order id returned for secondary order

ptAddBlockOrder

The ptAddBlockOrder 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.

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** (integer)	

Argument/Returns	Value	
PrimaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description.	
SecondaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd desc tion	
LegPrices	Address of a structure of type **LegPriceStruct**, see below.	
OrderIDs	Address of a structure of type **CrossingOrderIDs.** See below.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrPriceRequired*	Order type required a price and one was not provided.

Argument/Returns	Value	
	• *ptErrUnknownAccount*	Trader account does not match known trader.
	• *ptErrUnknownOrderType*	Order type is not known.
	• *ptErrUnknownContract*	Contract name/ date does not refer to a valid contract.
	• *ptErrTASUnavailable*	Transaction server is not connected.
	• *ptErrMDSUnavailable*	PDD is not connected.

LegPriceStruct

Field	Description
Leg0	string[21] – Leg0 price
Leg1 **…** **Leg14**	string[21] – Leg Prices
Leg15	string[21] - Leg15 price

CrossingOrderIDs

Field	Description
PrimaryOrderID	string[11] – order id returned for primary order
SecondaryOrderID	string[11] – order id returned for secondary order

ptAddCrossingOrder

The ptAddCrossingOrder 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.

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, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
PrimaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd description.	
SecondaryOrder	Address of a structure of type **NewOrderStruct** containing the order details. Refer to ptAddOrder for NewOrderAdd desc tion	
LegPrices	Address of a structure of type **LegPriceStruct**, see below.	
OrderIDs	Address of a structure of type **CrossingOrderIDs.** See below.	
FAK	char default 'L' (Leave).	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrPriceRequired*	Order type required a price and one was not provided.
	• *ptErrUnknownAccount*	Trader account does not match known trader.
	• *ptErrUnknownOrderType*	Order type is not known.
	• *ptErrUnknownContract*	Contract name/ date does not refer to a valid contract.
	• *ptErrTASUnavailable*	Transaction server is not connected.
	• *ptErrMDSUnavailable*	PDD is not connected.

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.

LegPriceStruct

Field	Description
Leg0	string[21] – Leg0 price
Leg1 **…** **Leg14**	string[21] – Leg Prices
Leg15	string[21] - Leg15 price

CrossingOrderIDs

Field	Description
PrimaryOrderID	string[11] – order id returned for primary order
SecondaryOrderID	string[11] – order id returned for secondary order

ptAddOrder

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.

Arguments:	• **NewOrder**	(struct read-only, by reference)
	• **OrderID**	(sting[11] writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
NewOrder	Address of a structure of type **NewOrderStruct** containing the order details. See below.	
OrderID	Address of a string[11] variable which will receive the OrderID of the order.	
Status	• *ptSuccess*	Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrPriceRequired*	Order type required a price and one was not provided.
	• *ptErrUnknownAccount*	Trader account does not match known trader.
	• *ptErrUnknownOrderType*	Order type is not known.
	• *ptErrUnknownContract*	Contract name/ date does not refer to a valid contract.
	• *ptErrTASUnavailable*	Transaction server is not connected.
	• *ptErrMDSUnavailable*	PDD is not connected.

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.

NewOrderStruct

Field	Description
TraderAccount	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	String[11] - variable containing the order type. This is one of the values returned by *ptGetOrderType*.
ExchangeName	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	String[11] - variable containing the contract name for the order. This is one of the *ContractName* values returned by *ptGetContract*.

ContractDate	String[51] - variable containing the contract date of the contract, as returned by *ptGetContract*.
BuyOrSell	char - variable either 'B' or 'S'.
Price	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	String[21]- variable containing a second price if required. For example, a limit price for a stop/limit order. Blank if not required.
Lots	Integer - variable containing the number of lots for the order. May be any positive integer.
LinkedOrder	String[11] - Not in current use
OpenOrClose	Char - variable either 'O' or 'C' indicating if the Order is to open or close the traders position.
Xref	Integer - An integer variable containing a user supplied cross-reference number. This cross-reference is no longer valid if the API is exited.
XrefP	Integer - variable containing a user supplied cross reference number. This cross-reference persists even if the API is shutdown
GoodTillDate	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	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	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	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	Integer - variable to set priority that can be sent along with the order. Has meaning only for some exchanges.
TriggerDate	String[9] - variable containing a date to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
TriggerTime	String[7] - variable containing a time to trigger the order. Has meaning for SyOMS order types and some other liquidity pools.
BatchID	String[11] - variable used to determine orders placed/received in batches (for example, wholesale trade orders).
BatchType	String[11] - variable that describes the type of batch being placed/received (for example, a crossing batch is type 42)
BatchCount	Integer - the number of records in the batch.
BatchStatus	String[11] - variable that describes the status of the batch as it is passed through the system.
ParentID	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	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	String[256] - reference variable that will be echoed back on subsequent order responses.
SenderLocationID	String[33] – the geographic location of end user (2 character country code). For users based in US and Canada this should also include state code: e.g. **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	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	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	String[21] - variable used to describe the ITM code assigned to the third party developer to comply with Connect 9.0 exchanges.
YDSPAudit	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	String[11] - The calculated Near Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
ICSFarLegPrice	String[11] - The calculated Far Leg Price required when trading an ICS expiry as part of the Connect 9.0 requirements
MinClipSize	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	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	Char - used to determine if the Clip size is to be randomly generated when the ghost order is working
TicketType	String[3] - used to describe the type of ticket used when the ghost order was placed. Patsystemst GUI specific
TicketVersion	String[4] - used to determine the version of ticket used when twuihe ghost order was placed. Patsystemst GUI specific
ExchangeField	String[11] - Patsystems specific field: please ignore
BOFID	String[21] - Patsystems specific field: please ignore

Badge	String[6] - Patsystems specific field: please ignore
LocalUserName	String[11] - Patsystems specific field: please ignore
LocalTrader	String[21] - Patsystems specific field: please ignore
LocalBOF	String[21] - Patsystems specific field: please ignore
LocalOrderID	String[11] - Patsystems specific field: please ignore
LocalExAcct	String[11] - Patsystems specific field: please ignore
RoutingID1	String[11] - Patsystems specific field: please ignore
RoutingID2	String[11] - Patsystems specific field: please ignore
Inactive	Char -
clientIdShortCode	String[21] – exchange short code
clientIdType	Char – valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
commodityDerInd	Char - valid values: Y – Yes N - No
DEA	Char – valid values: Y – Yes N - No
executionDecision	String[21] – exchange short code
executionDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
investmentDecision	String[21] – exchange short code
investmentDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
liquidityProvider	Char - valid values: Y – Yes N - No

shortSelling	Char - valid values: 0 – <empty> 1 – SESH – no exception 2 – SESX – with exception 3 – SELL – no short sale 4 – UNDI – not available
tradingCapacity	Char - valid values: 1– DEAL 2 – MTCH 3 - AOTC
ancillaryTrading	Char - valid values: Y – Yes N - No

ptAddOrderEx

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.

Arguments:	• **NewOrder**	(struct read-only, by reference)
	• **OrderID**	(string[11] writeable, by reference)
	• **UserName**	(string[11] writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
NewOrder	Address of a structure of type **NewOrderStruct** containing the order details. See ptAddOrder.
OrderID	Address of a string[11] variable which will receive the OrderID of the order.

Argument/Returns	Value	
Username	A string[11] variable to receive the username to attach to the trade.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrPriceRequired*	Order type required a price and one was not provided.
	• *ptErrUnknownAccount*	Trader account does not match known trader.
	• *ptErrUnknownOrderType*	Order type is not known.
	• *ptErrUnknownContract*	Contract name/ date does not refer to a valid contract.
	• *ptErrTASUnavailable*	Transaction server is not connected.
	• *ptErrMDSUnavailable*	PDD is not connected.

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 behavior or revocation of your license.

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

ptAddAlgoOrder

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.

Arguments:	• **NewOrder**	(struct read-only, by reference)
	• **BuffSize**	(integer read-only, by reference)
	• **AlgoBuff**	(struct read-only, by reference)
	• **OrderID**	(string[11] writeable, by reference)

Returns:	**status** (integer)

Argument/Returns	Value
NewOrder	Address of a structure of type **NewOrderStruct** containing the order details. See ptAddOrder.
BuffSize	Size of the AlgoBuff
AlgoBuff	Array of char containing the Algo XML.
OrderID	Address of a string[11] variable which will receive the OrderID of the order.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrPriceRequired* Order type required a price and one was not provided. • *ptErrUnknownAccount* Trader account does not match known trader. • *ptErrUnknownOrderType* Order type is not known. • *ptErrUnknownContract* Contract name/ date does not refer to a valid contract. • *ptErrTASUnavailable* Transaction server is not connected. • *ptErrMDSUnavailable* PDD is not connected. • *ptErrInvalidAlgoXML* This Algo Order contains incorrect XML information

ptAddProtectionOrder

This method is used to place 'Bracket' orders.

Arguments:	• **NewOrder**	(struct read-only, by reference)
	• **ProtectionOrder**	(struct read-only, by reference)

	• **OrderID**	(string[11] writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
NewOrder	Address of a structure of type **NewOrderStruct** containing the order details. See ptAddOrder.
ProtectionOrder	**ProtectionStruc** (see below) - Contains the additional information required by SyOMS to place the orders to protect the position taken by the original order (defined in NewOrderStruct).
OrderID	Address of a string[11] variable which will receive the OrderID of the order.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrPriceRequired* Order type required a price and one was not provided. • *ptErrUnknownAccount* Trader account does not match known trader. • *ptErrUnknownOrderType* Order type is not known. • *ptErrUnknownContract* Contract name/ date does not refer to a valid contract. • *ptErrTASUnavailable* Transaction server is not connected. • *ptErrMDSUnavailable* PDD is not connected.

ProtectionStruc

Field	Description
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
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

ptAmendOrder

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.

Arguments:	• **OrderID**	(string[11] writeable, by reference)
	• **NewDetails**	(struct read-only, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
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. See below.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrNoData* — API does not currently hold any order information. • *ptErrAmendDisabled* — Amend is not supported by the exchange. Use cancel/add. • *ptErrInvalidState* — Order may not be amended at this time. • *ptErrInvalidPrice* — New price not valid. • *ptErrInvalidVolume* — New volume not valid. • *ptErrUnknownAccount* — Contract name/ date does not refer to a valid contract. • *ptErInvalidAmendOrderType* — *Order cannot be amended to this order type*.

OrderAmendStruct

Field	Description
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: e.g. 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
clientIdShortCode	String[21] – exchange short code
clientIdType	Char – valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
commodityDerInd	Char - valid values: Y – Yes N - No
DEA	Char - valid values: Y – Yes N - No
executionDecision	String[21] – exchange short code
executionDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
investmentDecision	String[21] – exchange short code
investmentDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
liquidityProvider	Char - valid values: Y – Yes N - No
shortSelling	Char - valid values: 0 – <empty> 1 – SESH – no exception 2 – SESX – with exception

	3 – SELL – no short sale 4 – UNDI – not available
tradingCapacity	Char - valid values: 1– DEAL 2 – MTCH 3 - AOTC
ancillaryTrading	Char - valid values: Y – Yes N - No

ptAmendOrderEx

The normal call (ptAmendOrder) attaches the logon username to the trade. The ptAmendOrderEx 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. 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.

Arguments:	• **OrderID**	(string[11] writeable, by reference)
	• **NewDetails**	(struct read-only, by reference)
	• **UserName**	(string[11] read-only, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
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. See *ptAmendOrder*.	
Username	A string[11] variable to receive the username to attach to the trade.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)

Argument/Returns	Value	
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	API does not currently hold any order information.
	• *ptErrAmendDisabled*	Amend is not supported by the exchange. Use cancel/add.
	• *ptErrInvalidState*	Order may not be amended at this time.
	• *ptErrInvalidPrice*	New price not valid.
	• *ptErrInvalidVolume*	New volume not valid.
	• *ptErrUnknownAccount*	Contract name/ date does not refer to a valid contract.
	• *ptErInvalidAmendOrderType*	*Order cannot be amended to this order type*.

ptAmendAlgoOrder

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.

Arguments:	• **OrderID**	(string[11] writeable, by reference)
	• **BuffSize**	(integer read-only, by reference)
	• **AlgoBuff**	(struct read-only, by reference)
	• **NewDetails**	(struct read-only, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
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.
BuffSize	Size of the AlgoBuff

Argument/Returns	Value
AlgoBuff	Array of char containing the Algo XML.
NewDetails	Address of a structure of type **AmendOrderStruct**, containing the new details for the order. See ptAmendOrder.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* API does not currently hold any order information. • *ptErrAmendDisabled* Amend is not supported by the exchange. Use cancel/add. • *ptErrInvalidState* Order may not be amended at this time. • *ptErrInvalidPrice* New price not valid. • *ptErrInvalidVolume* New volume not valid. • *ptErrUnknownAccount* Contract name/ date does not refer to a valid contract. • *ptErInvalidAmendOrderType* *Order cannot be amended to this order type*.

ptAtBest (callback)

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*.

Arguments:	• **AtBestData** (struct read-only, by reference)
Returns:	**None**

Argument/Returns	Value
AtBestData	Address of a structure of type **AtBestUpdStruct**. The application routine will receive the contract updated in this parameter. **AtBestUpdStruct** **read-only, by reference** ExchangeName: string[11] ContractName: string[11] ContractDate: string[11]

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.

Note: The callback must be registered with the *ptRegisterAtBestCallback* routine.

ptBlankPrices

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.

Arguments:	**None**
Returns:	**None**

ptCancelOrder

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.

Arguments:	• **OrderID** (string[11] read-only, by reference) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
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.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrNoData* — The API holds no order information. • *ptErrUnknownOrder* — The order ID does not refer to a known order. • *ptErrInvalidState* — Order is not in a valid state to cancel.

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.

ptCancelOrderEx

The ptCancelOrderEx routine is functionally the same ptCancelOrder, submits a cancellation for the specified order. However additional arguments are available.

Arguments:	• **OrderID**	(string[11]	read-only, by reference)
	• **UserName**	(string[11]	read-only, by reference)

	• **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
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.
UserName	Address of a string[11] variable containing the username to be used for the action.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrNoData* — The API holds no order information. • *ptErrUnknownOrder* — The order ID does not refer to a known order. • *ptErrInvalidState* — Order is not in a valid state to cancel.

ptCancelOrderEx2

The ptCancelOrderEx2 routine is functionally the same ptCancelOrder, submits a cancellation for the specified order. However additional arguments are available.

Arguments:	• **OrderID** (string[11] read-only, by reference) • **ClientID** (string[21] read-only, by reference)

	• **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
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.
ClientID	Address of a string[21] variable containing the Patsystem ClientID value to be used.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information. • *ptErrUnknownOrder* The order ID does not refer to a known order. • *ptErrInvalidState* Order is not in a valid state to cancel.

ptCancelAll

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).

Arguments:	• **TraderAccount** (string[11] read-only, by reference)

	• **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
TraderAccount	Address of a string[21] variable containing the trader account to cancel orders for.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information.

ptCancelAllEx

The ptCancelAllEx 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).

Arguments:	• **TraderAccount** (string[11] read-only, by reference) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
TraderAccount	Address of a string[21] variable containing the trader account to cancel orders for.

Argument/Returns	Value
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information.

ptCancelBuys

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).

Arguments:	• **TraderAccount** (string[11] 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) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
TraderAccount	Address of a string[21] variable containing the trader account to cancel orders for.

Argument/Returns	Value
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.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information.

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)

ptCancelSells

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).

Arguments:	• **TraderAccount** (string[11] 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) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
TraderAccount	Address of a string[21] variable containing the trader account to cancel orders for.
ExchangeName	Address of a string[11] variable containing the exchange name of the contract to cancel orders for.
ContractName	Address of a string[11] variable containing the contract name to cancel orders for.
ContractDate	Address of a string[51] variable containing the contract date to delete orders for.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information.

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).

ptCancelOrders

The ptCancelOrders routine submits cancellations for all 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).

Arguments:	• **TraderAccount** (string[11] 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) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
TraderAccount	Address of a string[21] variable containing the trader account to cancel 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.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision

Argument/Returns	Value
	investmentDecision clientIdShortCode
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API holds no order information.

If the application has made a call to ptSetUserIDFilter to enable filtering, a call to ptCancelOrders will only cancel buy orders that have been entered by the currently logged in user.

ptActivateOrder

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.

Arguments:	• **OrderID** (string[11] read-only, by reference) • **OrderDetail** (struct read-only, by reference Optional)
Returns:	**status** (integer)

Argument/Returns	Value
OrderID	Address of a string[11] variable containing the **Patsystems** order ID of the order to be activated. This is the value returned by the *ptOrder* callback for a new order and uniquely identifies the order on PATS.
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode

Argument/Returns	Value	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	The API holds no order information.
	• *ptErrUnknownOrder*	The order ID does not refer to a known order.
	• *ptErrInvalidState*	Order is not in a valid state to activate.

ptDeactivateOrder

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.

Arguments:	• **OrderID**	(string[11]	read-only, by reference)
	• **OrderDetail**	(struct	read-only, by reference Optional)
Returns:	**status** (integer)		

Argument/Returns	Value	
OrderID	Address of a string[11] variable containing the **Patsystems** order ID of the order to be deactivated. This is the value returned by the *ptOrder* callback for a new order and uniquely identifies the order on PATS.	
OrderDetail (Optional)	Address of a structure of type OrderDetailStruct where the API will write the result. See ptGetOrder. Applicable fields to set/modify in the structure: DEA executionDecision investmentDecision clientIdShortCode	
Status	• *ptSuccess*	Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	The API holds no order information.
	• *ptErrUnknownOrder*	The order ID does not refer to a known order.
	• *ptErrInvalidState*	Order is not in a valid state to deactivate.

ptCountFills

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.

Arguments:	• **Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Count	Address of an integer variable where the API will write the result.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErNoData*	The API holds no fill information.

ptCountOrderHistory

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.

Arguments:	• **Index** (integer read-only, immediate value) • **Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
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.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Index value does not refer to a valid record.
	• *ptErNoData*	The API holds no order information.

ptCountOrders

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.

Arguments:	• **Count** (integer writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value	
Count	Address of an integer variable where the API will write the result.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.

Argument/Returns	Value	
	• *ptErNoData*	The API holds no order information.

ptCountContractAtBest

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*.

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** (integer)	

Argument/Returns	Value	
ExchangeName	Address of a string[11] variable containing the exchange name.	
ContractName	Address of a string[11] variable containing the contract name.	
ContractDate	Address of a string[51] variable containing the contract date.	
Count	Address of an integer variable where the API will write the result.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrUnknownContract*	Exchange, Contract and Date not recognised.

ptCountContractSubscriberDepth

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*.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
ContractName	Address of a string[11] variable containing the contract name.
ContractDate	Address of a string[51] variable containing the contract.
Count	Address of an integer variable where the API will write the result.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

Subscriber Depth is like 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 (ASX).

ptFill (callback)

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.

Arguments:	• **OrderID** (string[11] read-only, by reference) • **FillID** (string[71] read-only, by reference)
Returns:	**None**

Argument/Returns	Value
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.

Note: 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* 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.

ptGetAveragePrice

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.

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** (integer)	

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* API does not hold any fill data at the current time. • *ptErrUnknownContract* Exchange, Contract and Date not recognised. • *ptErrNotEnabled* This function is not enabled for gateway application.

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.

ptGetContractAtBest

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.

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) • **AtBestData** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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*.
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.
AtBestData	Address of a data structure of type **AtBestStruct** where the API will write the details. The structure is defined as: **AtBestStruct read-only, by reference** Firm: string[4] Volume: integer BestType: char - 'B' if the At Best price is a bid or 'O' if the At Best price is an offer.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrInvalidIndex* Value of index is out of range. • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

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.

ptGetContractAtBestPrices

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.

Arguments:	• **ExchangeName** (string[11] read-only, by reference) • **ContractName** (string[11] read-only, by reference) • **ContractDate** (string[51] read-only, by reference) • **AtBestPrices** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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*.
AtBestPrices	Address of a data structure of type **AtBestPricesStruct** where the API will write the price details. The structure is defined as: **AtBestPricesStruct** **read-only, by reference** BidPrice: string[21] OfferPrice: string[21] LastBuyer: string[4] Lastseller: string[4]
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host.

Argument/Returns	Value
	• *ptErrUnknownContract* Exchange, Contract and Date not recognised.

ptGetContractPosition

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.

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** (integer)

Argument/Returns	Value
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: **PositionStruct** Profit: string[21] - the total profit in the contract, reported in contract currency. Converts to a

Argument/Returns	Value
	floating point number. Buy: integer - the current total buy volume in this contract. Sell: integer - the current total sell volume in this contract.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* API does not hold any fill data at the current time. • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

ptGetContractSubscriberDepth

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.

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) • **SubscriberDepthData** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
ContractName	Address of a string[11] variable containing the contract name to query. This value is one of the values returned by *ptGetContract*.

Argument/Returns	Value
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*.
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.
SubscriberDepthData	Address of a data structure of type **SubscriberDepthStruct** where the API will write the price details. The structure is defined as: **SubscriberDepthStruct** **read-only, by reference** Price: string[21] Volume: integer Firm: string[4] DepthType char - that contains B for bid or O for offer
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrInvalidIndex* Value of index is out of range. • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

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.

ptGetFill

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, to obtain all fills for an order, the entire list of fills is read and fills for other orders ignored.

Arguments:	• **Index**	(integer read-only, by reference)
	• **FillDetails**	(struct writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
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. See below.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Value of index is out of range.
	• *ptErrNoData*	API does not hold any fill data at this time.

FillStruct

Field	Description
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 Trading 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
clientIdShortCode	String[21] – exchange short code

clientIdType	Char – valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
commodityDerInd	Char - valid values: Y – Yes N - No
DEA	Char - valid values: Y – Yes N - No
executionDecision	String[21] – exchange short code
executionDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
investmentDecision	String[21] – exchange short code
investmentDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
liquidityProvider	Char - valid values: Y – Yes N - No
shortSelling	Char - valid values: 0 – <empty> 1 – SESH – no exception 2 – SESX – with exception 3 – SELL – no short sale 4 – UNDI – not available
tradingCapacity	Char - valid values: 1– DEAL 2 – MTCH 3 - AOTC
ancillaryTrading	Char - valid values: Y – Yes N - No
fillTimeStamp	Time of fill - YYYYMMDD:hh.mm.ss.ssssss

ptGetFillByID

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.

Arguments:	• **FillID**	(string[71] read-only, by reference)
	• **FillDetails**	(struct writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
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*.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Value of index is out of range.
	• *ptErrNoData*	API does not hold any fill data at this time.

ptGetGenericPrice

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.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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: **PriceDetailStruct** Price: string[21] Volume: integer - the volume (if relevant) of the price type requested AgeCounter: byte – if zero the price has expired Direction: byte – 0 = same, 1 = rise, 2 = fall Hour: byte Minute: byte Second: byte
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

ptGetOpenPosition

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.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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: **PositionStruct** Profit: string[21] - the total profit in the contract, reported in contract currency. Converts to a floating point number. Buy: integer - current open buy volume. Sell: integer - current open sell volume.
Status	• *ptSuccess* Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	The API does not hold any fill data time
	• *ptErrUnknownContract*	Exchange, Contract and Date not recognised.

ptGetOrder

The ptGetOrder routine returns the details for an order held for the user in the API. The data is returned in Order ID order.

Arguments:	• **Index**	(integer read-only, by reference)
	• **Orderdetail**	(struct writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
Index	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. See below.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Value of index is out of range.
	• *ptErrNoData*	API does not hold any fill data at this time.

OrderDetailStruct

Field	Description
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: e.g. 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
clientIdShortCode	String[21] – exchange short code
clientIdType	Char – valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
commodityDerInd	Char - valid values: Y – Yes N - No
DEA	Char - valid values: Y – Yes N - No

executionDecision	String[21] – exchange short code
executionDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
investmentDecision	String[21] – exchange short code
investmentDecisionType	Char - valid values: 0 – Undef 1 – Person 2 – Algorithm 3 – Legal Entity
liquidityProvider	Char - valid values: Y – Yes N - No
shortSelling	Char - valid values: 0 – <empty> 1 – SESH – no exception 2 – SESX – with exception 3 – SELL – no short sale 4 – UNDI – not available
tradingCapacity	Char - valid values: 1– DEAL 2 – MTCH 3 - AOTC
ancillaryTrading	Char - valid values: Y – Yes N - No

Order states

State	Description
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.

Order sub-states

State	Description
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

Fill sub-states

State	Description
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

ptGetOrderEx

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.

Arguments:	• **Index**	(integer read-only, by reference)
	• **AlgoDetail**	(struct writeable, by reference)
	• **AlgoSize**	(integer writeable, by reference)
	• **OrderDetail**	(struct writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
Index	Integer value indicating which record to return. Supply a value between 0 and ptCountOrders – 1.
AlgoDetail	Array of char containing the Algo XML.
AlgoSize	Size of the AlgoDetail

Argument/Returns	Value
OrderDetail	Address of a structure of type **OrderDetailStruct** where the API will write the result. See below.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrInvalidIndex* Value of index is out of range. • *ptErrNoData* API does not hold any fill data at this time.

ptGetOrderByID

The ptGetOrderById routine returns the details for an order held for the user in the API. The data is returned in chronological order.

Arguments:	• **OrderID** (string[11] read-only, by reference) • **OrderDetail** (struct writeable, by reference) • **OFSequence** (integer read-only, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful

Argument/Returns	Value	
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Value of index is out of range.
	• *ptErrNoData*	API does not hold any fill data at this time.

ptGetOrderByIDEx

The ptGetOrderByIdEx 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.

Arguments:	• **OrderID**	(string[11] read-only, by reference)
	• **OrderDetail**	(struct writeable, by reference)
	• **AlgoDetail**	(struct writeable, by reference)
	• **AlgoSize**	(integer writeable, by reference)
	• **OFSequence**	(integer read-only, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value
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.
AlgoDetail	Array of char containing the Algo XML.
AlgoSize	Size of the AlgoDetail
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

Argument/Returns	Value
	backwards compatibility. This value is passed back to the client in the Order Callback.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrInvalidIndex* Value of index is out of range. • *ptErrNoData* API does not hold any fill data at this time.

ptGetOrderHistory

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).

Arguments:	• **Index** (integer read-only, by reference) • **Position** (integer read-only, by reference) • **OrderDetail** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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 details of OrderDetailStruct.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrInvalidIndex* Value of index is out of range.

Argument/Returns	Value	
	• *ptErrNoData*	API does not hold any fill data at this time.

ptGetOrderHistoryEx

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.

Arguments:	• **Index**	(integer read-only, by reference)
	• **Position**	(integer read-only, by reference)
	• **OrderDetail**	(struct writeable, by reference)
	• **AlgoDetail**	(struct writeable, by reference)
	• **AlgoSize**	(integer writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
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 details of OrderDetailStruct.	
AlgoDetail	Array of char containing the Algo XML.	
AlgoSize	Size of the AlgoDetail	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrInvalidIndex*	Value of index is out of range.

Argument/Returns	Value	
	• *ptErrNoData*	API does not hold any fill data at this time.

ptGetPrice

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.

Arguments:	• **Index**	(integer read-only, by reference)
	• **CurrentPrice**	(struct writeable, by reference)
Returns:	**status** (integer)	

Argument/Returns	Value	
Index	Address of a string[11] variable containing the exchange name.	
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 PriceDetailStruct, 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. See below.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	Exchange, Contract and Date not recognised.
	• *ptErrInvalidIndex*	Value of index is out of range.

PriceDetailStruct

Field	Description
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.

PriceDetailStruct

Note: There are 20 levels of Depth of Market data but not all are always populated. There are 20 levels of Last Traded data for providing time and sales.

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
Last1 … Last18	PriceDetailStruct	
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)
BidDOM1 … BidDOM18	PriceDetailStruct	

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	Integer	An integer containing the current Market Status of the contract date. See ptStatusChange for details.
Mask	Integer	An integer bitmask indicating which prices have changed since the last call to ptGetPrice or ptGetPriceForContract.
PriceStatus	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.

Price

Type	Description
ptChangeBid	Offer Price/Volume has changed
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

ptGetPriceForContract

The ptGetPriceForContract routine returns price information for a contract. This call is similar in nature to ptGetPrice.

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)
Returns:	**status** (integer)

Argument/Returns	Value
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

Argument/Returns	Value
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 PriceDetailStruct, 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. See ptGetPrice.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* Exchange, Contract and Date not recognised. • *ptErrInvalidIndex* Value of index is out of range.

ptGetTotalPosition

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.

Arguments:	• **TraderAccount** (string[21] read-only, by reference) • **Position** (struct writeable, by reference)
Returns:	**status** (integer)

Argument/Returns	Value
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: **PositionStruct** Profit: string[21] - the total profit in the contract, reported in contract currency. Converts to a floating point number.

Argument/Returns	Value
	Buy: integer - current open buy volume. Sell: integer - current open sell volume.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* The API does not hold any fill data time • *ptErrUnknownContract* Exchange, Contract and Date not recognised.

ptOrder (callback)

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.

Arguments:	• **OrderData** (struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
OrderData	Address of a structure of type O**rderUpdStruct**. The application routine will receive the order details in this parameter. **OrderUpdStruct** OrderID: string[11] – order id OldOrderID: string[11] - 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: byte - the current status of the order. OFSeqNumber: integer - the order sequence number OderTypeID: integer - 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.

Note: The callback must be registered with the *ptRegisterOrderCallback* routine.

ptOrderChecked

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.

Arguments:	• **OrderID**	(string[11]	read-only, by reference)
	• **Checked**	(char	read-only, by reference)
Returns:	**status** (integer)		

Argument/Returns	Value	
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.	
Checked	Value to be set by the application	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrUnknownOrder*	Order specified does not exist.
	• *ptErrNoData*	API does not hold any fill data at this time.

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.

ptGetPriceSnapShot

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.

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** (integer)

Argument/Returns	Value
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
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrNoData* Exchange, Contract and Date not recognised.

Argument/Returns	Value	
	• *ptErrUnknownContract*	The Exchange, Contract and Date was not recognised

ptGetPriceStep

The ptPriceStep routine can be used to adjust a price up or down by a number of ticks.

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:	**adjustedPrice** (double)	

Argument/Returns	Value
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.
adjustedPrice	New adjusted price.

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.

ptPriceUpdate (callback)

The ptPriceUpdate callback fires whenever a new price is received for any contract. The callback returns the contract for which the price has changed.

Arguments:	• **PriceData**	**(**struct writeable, by reference)
Returns:	**None**	

Argument/Returns	Value
PriceData	Address of a structure of type **PriceUpdStruct**. The application routine will receive the order details in this parameter. **PriceUpdStruct** ExchangeName: string[11] ContractName: string[11] ContractDate: string[51]

Note: The callback must be registered with the *ptRegisterPriceCallback* routine.

ptPurge

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.

Arguments:	• **PDate** (string read-only, by reference) • **PTime** (string read-only, by reference)
Returns:	**None**

Argument/Returns	Value
Date	
Time	

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.

ptQueryOrderStatus

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.

Arguments:	• **OrderID** (string[11] read-only, by reference**)**
Returns:	**status** (integer)

Argument/Returns	Value	
OrderID	Address of a string[11] variable containing the **Patsystems** Order ID that identifies this order on the system.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.
	• *ptErrNoData*	Exchange, Contract and Date not recognised.
	• *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.

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.

ptSetUserIDFilter

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.*

Arguments:	• **Enable** **(**char readonly, immediate value)
Returns:	**status** (integer)

Argument/Returns	Value
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.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise)

ptStatusChange (callback)

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.

Arguments:	• **Data** (struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
Data	Address of a structure of type **StatusUpdStruct** containing details about what status changed, and its new value. See below.

Note: The callback must be registered with the *ptRegisterStatusCallback* routine.

StatusUpdStruct

Field	Description
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 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.

Status Values

Status	Description
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

ptSubscriberDepthUpdate (callback)

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.

Arguments:	• **SubscriberDepthData** (struct 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:	**None**

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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*.
SubscriberDepthData	Address of a data structure of type **SubscriberDepthStruct** where the API will write the price details. The structure is defined as:

Argument/Returns	Value
	SubscriberDepthStruct **read-only, by reference** Price: string[21] Volume: integer Firm: string[4] DepthType char - that contains B for bid or O for offer

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.

Note: The callback must be registered with the p*tRegisterSubscriberDepthCallback* routine.

ptTicker (callback)

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.

Arguments:	• **Data** **(**struct writeable, by reference)
Returns:	**None**

Argument/Returns	Value
Data	Address of a structure of type **TickerUpdStruct** containing details about what ticker change, and its new value. See below.

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 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.

Note: The callback must be registered with the *ptRegisterTickerCallback* routine.

TickerUpdStruct

Field	Description
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: 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.

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	the margin-per-lot required to trade the specific contract
Vol	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 each contract.

Buying Power Remaining

Buying Power Remaining is expressed as:

BPRemaining = SODLNV – OPE + P&L

Where:

SODLNV	the start of day net liquidity value, loaded from the backoffice
OPE	the open position exposure
P&L	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.

ptBuyingPowerRemaining

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.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.
BPRemaining	Address of a string[21] variable containing the value for the Buying Power Remaining as a percentage of the SODLNV+P&L.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host.

Argument/Returns	Value	
	• *ptErrUnknownAccount*	Supplied TraderAccount name does not refer to a valid record.

ptBuyingPowerUsed

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.

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** (integer)	

Argument/Returns	Value	
ExchangeName	Address of a string[11] variable containing the exchange name.	
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.	
BPUsed	Address of a string[21] variable containing the value for the Buying Power Used as a percentage of the SODLNV+P&L.	
Status	• *ptSuccess*	Successful
	• *ptErrNotInitilaised*	API is not initialised (ptInitialise)
	• *ptErrNotLoggedOn*	The API is not currently logged on to the host.

Argument/Returns	Value	
	• *ptErrUnknownAccount*	Supplied TraderAccount name does not refer to a valid record.

ptMarginForTrade

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.

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** (integer)		

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.

Argument/Returns	Value
Lots	Address of an integer variable containing the number of lots about to be traded.
OrderID	Address of a string[11] variable containing The ID of the order you are amending. If you are not amending an order leave this field blank.
Price	Address of a string[21] variable containing the price you are amending the order to, if you are not amending the order leave this field blank.
MarginReqd	Address of a string[21] variable containing the Margin required for the trade about to be made.
Status	• *ptSuccess* Successful • *ptErrNotInitilaised* API is not initialised (ptInitialise) • *ptErrNotLoggedOn* The API is not currently logged on to the host. • *ptErrUnknownAccount* Supplied TraderAccount name does not refer to a valid record.

ptOpenPositionExposure

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.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.

Argument/Returns	Value
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.
Exposure	Address of a string[21] variable containing the Open Position Exposure, expressed as a percentage of the total available "cash", SODNLV + P&L.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrUnknownAccount* — Supplied TraderAccount name does not refer to a valid record.

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:

$$\text{Open Position Exposure (\%)} = \frac{\text{No. of Lots} * \text{Margin}}{(\text{SODNLV} + \text{P\&L})} * 100$$

ptPLBurnRate

The ptPLBurnRate routine is used to retrieve the buying power burn rate for a trader account for a given Contract.

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** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.
BurnRate	Address of a string[21] variable to write the Burn Rate (as a percentage) to.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrUnknownAccount* — Supplied TraderAccount name does not refer to a valid record.

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 / SODNLV) * 100.

ptGetMarginPerLot

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.

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** (integer)	

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.
MarginReqd	Address of a string[21] variable containing the Margin required for the trade about to be made.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrUnknownAccount* — Supplied TraderAccount name does not refer to a valid record.

ptTotalMarginPaid

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.

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) • **TotalMargin** (string[21] writeable, by reference)
Returns:	**Status** (integer)

Argument/Returns	Value
ExchangeName	Address of a string[11] variable containing the exchange name.
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.
TotalMargin	Address of a string[21] variable containing the Total Margin for the trader account against the specified Contract date.
Status	• *ptSuccess* — Successful • *ptErrNotInitilaised* — API is not initialised (ptInitialise) • *ptErrNotLoggedOn* — The API is not currently logged on to the host. • *ptErrUnknownAccount* — Supplied TraderAccount name does not refer to a valid record.

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.