

OrderManagerUpdate Library Documentation

- [OrderManagerUpdate Library Documentation](#)
 - [Enumerations](#)
 - [EAState](#)
 - [RiskManagementMode](#)
 - [Usage](#)
 - [RequestStatus](#)
 - [Usage](#)
 - [Structures](#)
 - [TradeRequest](#)
 - [Usage](#)
 - [TicketRequest](#)
 - [Usage](#)
 - [Classes](#)
 - [COrderManager](#)
 - [COrderManager\(\)](#)
 - [Usage](#)
 - [Init\(\)](#)
 - [Usage](#)
 - [AddSymbol\(\)](#)
 - [Usage](#)
 - [RefreshSymbolInfo\(\)](#)
 - [Usage](#)
 - [SetRiskSettings\(\)](#)
 - [Usage](#)
 - [CalcRiskVolumeSymbol\(\)](#)
 - [Usage](#)
 - [Trade\(\)](#)
 - [Usage](#)
 - [ClosePosition\(\)](#)
 - [Usage](#)
 - [DeletePendingOrder\(\)](#)
 - [Usage](#)
 - [ModifyPosition\(\)](#)
 - [Usage](#)
 - [GetRequestStatus\(\)](#)
 - [Usage](#)
 - [IsBusy\(\)](#)
 - [Usage](#)
 - [Process\(\)](#)
 - [Usage](#)
 - [Usage Example](#)

Enumerations

EASState

Description: Enumeration that defines the different action states for ticket-based operations (positions and pending orders). Used internally to specify what action should be performed on a position or order ticket when processing the ticket queue.

Values:

- `STATE_PROCESSING_CLOSE` - Close an open position
 - `STATE_PROCESSING_DELETE` - Delete a pending order
 - `STATE_PROCESSING_MODIFY` - Modify the stop loss and take profit levels of an open position
-

RiskManagementMode

Description: Enumeration that defines the risk management mode for automatic volume calculation. Determines how the order manager calculates position size based on risk parameters.

Values:

- `RiskFixedLot` - Fixed lot size mode; use a constant volume regardless of risk settings
- `RiskPercentBalance` - Percentage of account balance mode; calculate volume based on a percentage of current balance and the stop loss distance
- `RiskFixedMoney` - Fixed money risk mode; calculate volume to risk a fixed amount of money per trade

Usage

```
// Fixed lot mode: Always trade 0.1 lots
orderMgr.SetRiskSettings(RiskFixedLot, 0.1);
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0, 0, 1.0700, 1.0900, "Fixed Lot", 1001);
// When vol=0, automatically uses 0.1 lots

// Percentage of balance mode: Risk 2% of account per trade
orderMgr.SetRiskSettings(RiskPercentBalance, 2.0);
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0, 0, 1.0700, 1.0900, "Risk 2%", 1002);
// When vol=0, calculates volume based on 2% of account balance and stop loss
distance

// Fixed money mode: Risk $50 per trade
orderMgr.SetRiskSettings(RiskFixedMoney, 50.0);
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0, 0, 1.0700, 1.0900, "Risk $50", 1003);
// When vol=0, calculates volume to risk exactly $50 based on stop loss distance
```

RequestStatus

Description: Enumeration that represents the current status of a trade request identified by its request ID. Used to track the lifecycle of submitted orders and ticket operations.

Values:

- `REQ_STATUS_SUCCESS` - Request successfully completed; not found in pending queues or fatal error buffer
- `REQ_STATUS_PENDING` - Request is still in the queue waiting to be processed or is currently retrying
- `REQ_STATUS_ERROR` - Request encountered a fatal error and is stored in the fatal error buffer

Usage

```
// Check status after submitting a trade
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0.1, 0, 1.0700, 1.0900, "Buy Trade",
1001);

// Later, check the request status
RequestStatus status = orderMgr.GetRequestStatus(1001);
if(status == REQ_STATUS_SUCCESS) {
    Print("Trade 1001 executed successfully");
}
else if(status == REQ_STATUS_PENDING) {
    Print("Trade 1001 is still processing or retrying");
}
else if(status == REQ_STATUS_ERROR) {
    Print("Trade 1001 encountered a fatal error");
}
```

Structures

TradeRequest

Description: Represents a single trade request to open a new position or pending order. Contains all necessary parameters for order execution including price, volume, and risk levels. This structure is used internally by the order manager's trade queue system.

Members:

Member	Type	Description
type	ENUM_ORDER_TYPE	The order type (e.g., ORDER_TYPE_BUY, ORDER_TYPE_SELL, ORDER_TYPE_BUY_LIMIT, ORDER_TYPE_SELL_STOP, etc.). Determines whether to open a market or pending order.
volume	double	The position size in lots. Must comply with the symbol's volume step, minimum, and maximum constraints. Automatically normalized by the order manager.
price	double	The entry price for the order. For market orders (BUY/SELL), set to 0 to use current market price. For pending orders, specify the trigger price. Automatically normalized to tick size.

Member	Type	Description
<code>sl</code>	<code>double</code>	Stop loss price in quote currency. Set to 0 if no stop loss is required. Must respect the symbol's minimum stop level distance from the entry price.
<code>tp</code>	<code>double</code>	Take profit price in quote currency. Set to 0 if no take profit is required. Automatically normalized to tick size.
<code>expiration</code>	<code>datetime</code>	Expiration time for pending orders using ORDER_TIME_SPECIFIED. Set to 0 for GTC (Good-Till-Canceled) orders.
<code>comment</code>	<code>string</code>	Order comment/label for identification and logging purposes. Limited to 31 characters in MT5.
<code>retryAt</code>	<code>datetime</code>	Internal field; timestamp when the request should be retried. Automatically managed by the order manager.
<code>requestID</code>	<code>int</code>	Unique identifier for tracking this request. Used for status queries and error reporting. Must be greater than 0.

Usage

```
// TradeRequest is created internally when calling Trade()
// Example: Submitting a market buy order
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0.1, 0, 1.0700, 1.0900, "Buy Signal",
2001);

// Example: Submitting a pending buy-stop order
double currentPrice = SymbolInfoDouble("EURUSD", SYMBOL_ASK);
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY_STOP, 0.1,
               currentPrice + 0.0050, currentPrice + 0.0020, currentPrice -
0.0030,
               "Breakout Buy", 2002);

// Example: Pending order with expiration
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY_LIMIT, 0.1, 1.0650, 1.0800, 1.0500,
               "Limit Buy", 2003, D'2026.02.10 17:00:00');
```

TicketRequest

Description: Represents a single ticket-based request to modify, close, or delete a position or pending order. Used for managing existing positions and orders through the order manager's ticket queue system.

Members:

Member	Type	Description
--------	------	-------------

Member	Type	Description
<code>ticket</code>	<code>ulong</code>	The ticket/order ID of the position or pending order to operate on. Must be a valid ticket number greater than 0.
<code>action</code>	<code>EASState</code>	The action to perform: <code>STATE_PROCESSING_CLOSE</code> (close position), <code>STATE_PROCESSING_DELETE</code> (delete pending order), or <code>STATE_PROCESSING_MODIFY</code> (modify stop loss/take profit).
<code>sl</code>	<code>double</code>	New stop loss price for <code>MODIFY</code> actions. Set to 0 if not modifying stop loss. Ignored for <code>CLOSE</code> and <code>DELETE</code> actions. Automatically normalized to tick size.
<code>tp</code>	<code>double</code>	New take profit price for <code>MODIFY</code> actions. Set to 0 if not modifying take profit. Ignored for <code>CLOSE</code> and <code>DELETE</code> actions. Automatically normalized to tick size.
<code>retryAt</code>	<code>datetime</code>	Internal field; timestamp when the request should be retried. Automatically managed by the order manager.
<code>requestID</code>	<code>int</code>	Unique identifier for tracking this request. Used for status queries and error reporting. Must be greater than 0.

Usage

```
// TicketRequest is created internally when calling ClosePosition(),
DeletePendingOrder(), or ModifyPosition()
// Example: Close a position
ulong positionTicket = 123456;
orderMgr.ClosePosition("EURUSD", positionTicket, 3001);

// Example: Delete a pending order
ulong pendingOrderTicket = 123457;
orderMgr.DeletePendingOrder("EURUSD", pendingOrderTicket, 3002);

// Example: Modify stop loss and take profit
orderMgr.ModifyPosition("EURUSD", positionTicket, 1.0650, 1.0950, 3003);
```

Classes

COrderManager

Description: A sophisticated order management class that handles multi-symbol trading with queued order execution, automatic retry logic, and risk management. It implements circular buffer queues for efficient order processing, supports up to 16 different symbols simultaneously, and provides request status tracking through request IDs. The class optimizes for both live trading (with processing budgets and spread checks) and backtesting (with relaxed timing constraints). All trade requests are normalized to symbol specifications before execution.

COrderManager()

Signature:

```
COrderManager()
```

Description: Constructor that initializes a new COrderManager instance. Sets up internal data structures including circular buffers for trade and ticket queues, initializes symbol tracking, and prepares the fatal error buffer. All arrays and counters are reset to zero. This must be called before any other methods.

Arguments: None

Returns: `void` - No return value. Initializes the object in memory.

Usage

```
COrderManager orderMgr; // Create instance
// Object is now ready for initialization via Init()
```

Init()**Signature:**

```
void Init(string symbol, int magic, int retryDelay=5, int maxSpread=50, int
slippage=10, uint timeBudgetMs=200)
```

Description: Initializes the order manager with the first symbol and trading parameters. Sets up the CTrade helper object with magic number and slippage, configures retry behavior, spread limits, and processing time budgets. In backtesting mode, timing constraints and retry delays are automatically disabled for maximum performance. This method must be called once before using Trade(), ClosePosition(), or other trading methods.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The name of the first symbol to trade (e.g., "EURUSD", "GBPUSD"). This symbol is registered immediately and symbol info is fetched.
<code>magic</code>	<code>int</code>	Magic number for the expert advisor. Used to identify trades opened by this EA in the terminal. Helps distinguish trades from different strategies.
<code>retryDelay</code>	<code>int</code>	Number of seconds to wait before retrying a failed request (default: 5). Set to 0 for no retry delay. Ignored in backtesting mode.

Parameter	Type	Description
<code>maxSpread</code>	<code>int</code>	Maximum acceptable spread in points before rejecting a market order (default: 50). If spread exceeds this value, the order waits for better conditions.
<code>slippage</code>	<code>int</code>	Maximum acceptable slippage in points for market orders (default: 10). Used when executing trades with the CTrade helper.
<code>timeBudgetMs</code>	<code>uint</code>	Maximum milliseconds per tick allowed for order processing (default: 200). In live trading, when exceeded, processing pauses until next tick. Ignored in backtesting mode.

Returns: `void` - No return value.

Usage

```
// Initialize with EURUSD as the primary symbol
orderMgr.Init("EURUSD", 12345); // Using default parameters

// Initialize with custom settings
orderMgr.Init("EURUSD", 12345, 5, 30, 5, 150);
// Retry after 5 seconds, max spread 30 points, slippage 5 points, 150ms budget
```

AddSymbol()

Signature:

```
bool AddSymbol(string symbol)
```

Description: Registers an additional symbol with the order manager for multi-symbol trading. Fetches symbol-specific information (tick size, volume constraints, stop levels, etc.) immediately upon registration. The order manager supports up to 16 symbols per instance. Attempting to add a duplicate symbol or exceeding the 16-symbol limit returns false without adding the symbol.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The name of the symbol to register (e.g., "AUDUSD", "GOLD"). Symbol name must be valid and exist on the broker's server.

Returns: `bool` - `true` if the symbol was successfully added, `false` if the symbol count would exceed 16 or the symbol is already registered.

Usage

```
// After Init(), add more symbols for multi-symbol trading
bool success = orderMgr.AddSymbol("GBPUSD");
if(success) Print("GBPUSD registered successfully");

// Add more symbols
orderMgr.AddSymbol("AUDUSD");
orderMgr.AddSymbol("NZDUSD");
orderMgr.AddSymbol("EURUSD"); // Already exists from Init(), this will fail
```

RefreshSymbolInfo()

Signature:

```
void RefreshSymbolInfo()
void RefreshSymbolInfo(int symbolIdx)
```

Description: Updates symbol-specific market information including tick size, points per unit, volume constraints, stop levels, and order filling type. The no-argument version refreshes all registered symbols. The indexed version refreshes a single symbol by its registration index. In live trading, symbol info is automatically refreshed every 10 seconds. In backtesting, refresh occurs on each new bar for efficiency. Manual calls to this function are rarely necessary.

Arguments (Overload 1 - No Parameters): None

Arguments (Overload 2 - Single Symbol):

Parameter	Type	Description
<code>symbolIdx</code>	<code>int</code>	Zero-based index of the symbol to refresh. The index corresponds to the order in which symbols were registered (first symbol = 0, second = 1, etc.). Must be between 0 and <code>symbolCount - 1</code> .

Returns: `void` - No return value.

Usage

```
// Refresh symbol info for all registered symbols
orderMgr.RefreshSymbolInfo();

// Refresh only a specific symbol (index 0 = first symbol)
orderMgr.RefreshSymbolInfo(0);

// Refresh second symbol
orderMgr.RefreshSymbolInfo(1);
```

SetRiskSettings()

Signature:

```
void SetRiskSettings(RiskManagementMode mode, double value)
```

Description: Configures the risk management parameters for automatic position sizing. When set, any `Trade()` call with `volume = 0` will automatically calculate or apply the position size based on these settings. For `RiskFixedLot` mode, the configured value is used directly. For `RiskPercentBalance` and `RiskFixedMoney` modes, the volume is calculated dynamically based on the order's stop loss distance. This enables consistent and intelligent position sizing across all trades.

Arguments:

Parameter	Type	Description
<code>mode</code>	<code>RiskManagementMode</code>	The risk calculation method: <code>RiskFixedLot</code> (fixed volume), <code>RiskPercentBalance</code> (percent of account balance), or <code>RiskFixedMoney</code> (fixed money amount).
<code>value</code>	<code>double</code>	The risk value interpreted based on the mode: Fixed lot size (e.g., 0.1), percentage of balance (e.g., 2.0 for 2%), or money amount in account currency (e.g., 50).

Returns: `void` - No return value.

Usage

```
// Configure fixed lot size: always trade 0.1 lots
orderMgr.SetRiskSettings(RiskFixedLot, 0.1);

// Configure percentage risk: risk 2% of account balance per trade
orderMgr.SetRiskSettings(RiskPercentBalance, 2.0);

// Configure fixed money risk: risk $50 per trade
orderMgr.SetRiskSettings(RiskFixedMoney, 50.0);

// Now Trade() with vol=0 will auto-calculate volume based on these settings
```

CalcRiskVolumeSymbol()

Signature:

```
double CalcRiskVolumeSymbol(string symbol, double entryPrice, double slPrice)
```

Description: Calculates the appropriate position size for a specific symbol based on the configured risk management mode, entry price, and stop loss price. If the risk mode is `RiskFixedLot`, returns the fixed volume directly. For percentage or fixed money modes, calculates the potential loss per lot and derives the position size. Returns the minimum volume if the calculation fails or if loss per lot is zero. This method is called automatically by `Trade()` when volume is 0, but can also be called manually for pre-calculation.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The symbol for which to calculate volume (e.g., "EURUSD"). Must be a registered symbol.
<code>entryPrice</code>	<code>double</code>	The entry price for the position in quote currency. Used to calculate the loss magnitude per lot.
<code>slPrice</code>	<code>double</code>	The stop loss price in quote currency. The distance between entry and stop loss determines the risk per lot.

Returns: `double` - The calculated position size in lots, automatically normalized to the symbol's volume step constraints and capped between minimum and maximum allowed volumes.

Usage

```
// Configure risk settings first
orderMgr.SetRiskSettings(RiskPercentBalance, 2.0);

// Calculate volume for a BUY trade with 2% risk
double entryPrice = 1.0850;
double stopLoss = 1.0800;
double volume = orderMgr.CalcRiskVolumeSymbol("EURUSD", entryPrice, stopLoss);
Print("Calculated volume: ", volume, " lots");

// Calculate volume for a SELL trade
volume = orderMgr.CalcRiskVolumeSymbol("EURUSD", 1.0850, 1.0900);
Print("Calculated volume: ", volume, " lots");
```

Trade()**Signature:**

```
bool Trade(string symbol, ENUM_ORDER_TYPE type, double vol, double price, double
sl, double tp, string comment, int requestID, datetime expiration=0)
```

Description: Submits a new trade request (market or pending order) to the trade queue for execution. If volume is 0, the order manager will automatically determine the position size based on the configured risk mode (see `SetRiskSettings()`). Market orders (BUY/SELL) are subject to spread checks and retry logic if spreads are too wide. All prices and volumes are normalized to symbol specifications. The request is queued and processed asynchronously. Use `GetRequestStatus()` with the requestID to check execution status.

Automatic Volume Behavior:

- If `vol = 0` and `RiskFixedLot` mode is configured: Uses the configured fixed lot value directly
- If `vol = 0` and `RiskPercentBalance` mode is configured: Calculates volume based on percentage of account balance and SL distance (requires `sl > 0`)
- If `vol = 0` and `RiskFixedMoney` mode is configured: Calculates volume to risk the fixed amount and SL distance (requires `sl > 0`)
- If `vol = 0` without any risk mode configured: Trade returns `false` (not queued)
- If `vol = 0` with `RiskPercentBalance` or `RiskFixedMoney` but `sl = 0`: Trade returns `false` with error message (cannot calculate without stop loss)

Important Notes:

- Always configure a risk mode via `SetRiskSettings()` before using `vol = 0` for automatic volume calculation
- For auto-volume calculation with `RiskPercentBalance` and `RiskFixedMoney` modes, a stop loss (`sl > 0`) is mandatory
- Market orders with `price = 0` use the current ASK for BUY orders and BID for SELL orders
- Pending orders require the exact trigger `price` to be specified
- The `expiration` parameter is only used for pending orders with `ORDER_TIME_SPECIFIED` mode

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The symbol to trade (e.g., "EURUSD"). Must be a registered symbol.
<code>type</code>	<code>ENUM_ORDER_TYPE</code>	The order type: <code>ORDER_TYPE_BUY</code> , <code>ORDER_TYPE_SELL</code> (market orders), <code>ORDER_TYPE_BUY_LIMIT</code> , <code>ORDER_TYPE_BUY_STOP</code> , <code>ORDER_TYPE_SELL_LIMIT</code> , <code>ORDER_TYPE_SELL_STOP</code> (pending orders).
<code>vol</code>	<code>double</code>	Position size in lots. Set to 0 to auto-calculate based on risk settings. Otherwise, must be between the symbol's minimum and maximum volume constraints.
<code>price</code>	<code>double</code>	Entry price for the order. For market orders (BUY/SELL), set to 0 to use current market price. For pending orders, specify the trigger price in quote currency.
<code>sl</code>	<code>double</code>	Stop loss price in quote currency. Set to 0 for no stop loss. Must respect the symbol's minimum stop level. Required if volume = 0 and auto-calculation is used.
<code>tp</code>	<code>double</code>	Take profit price in quote currency. Set to 0 for no take profit.

Parameter	Type	Description
<code>comment</code>	<code>string</code>	Order comment for identification and logging. Limited to 31 characters in MT5.
<code>requestID</code>	<code>int</code>	Unique identifier for this request (must be > 0). Used to track execution status via <code>GetRequestStatus()</code> .
<code>expiration</code>	<code>datetime</code>	Expiration time for pending orders using <code>ORDER_TIME_SPECIFIED</code> mode (default: 0 = GTC). Ignored for market orders.

Returns: `bool` - `true` if the request was successfully queued, `false` if:

- The symbol is not registered
- Volume is invalid (after normalization or auto-calculation, volume is ≤ 0)
- The queue is full (maximum 512 pending requests per symbol)
- Auto-volume calculation fails (e.g., `vol=0` with `RiskPercentBalance/RiskFixedMoney` but `sl=0`)
- No risk mode is configured and `vol=0` is provided

Usage

```
// Market BUY order with fixed volume
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0.1, 0, 1.0700, 1.0900, "Buy Signal",
1001);

// Market SELL order with automatic volume calculation (set vol=0)
orderMgr.SetRiskSettings(RiskPercentBalance, 2.0);
orderMgr.Trade("EURUSD", ORDER_TYPE_SELL, 0, 0, 1.0900, 1.0700, "Sell Signal",
1002);

// Buy-limit pending order
double limitPrice = SymbolInfoDouble("EURUSD", SYMBOL_ASK) - 0.0050;
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY_LIMIT, 0.1, limitPrice,
    limitPrice - 0.0050, limitPrice + 0.0100, "Limit Buy", 1003);

// Buy-stop pending order with expiration
double stopPrice = SymbolInfoDouble("EURUSD", SYMBOL_ASK) + 0.0050;
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY_STOP, 0.1, stopPrice,
    stopPrice - 0.0050, stopPrice + 0.0100, "Breakout", 1004,
D'2026.02.10 17:00');
```

ClosePosition()

Signature:

```
bool ClosePosition(string symbol, ulong ticket, int requestID)
```

Description: Submits a request to close an open position. The close request is queued and executed asynchronously. Before execution, the order manager verifies that the ticket exists and points to an open position. Use `GetRequestStatus()` with the `requestID` to check if the close was successful or is still pending.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The symbol of the position to close. Must be a registered symbol.
<code>ticket</code>	<code>ulong</code>	The ticket/order ID of the open position to close. Must be a valid, non-zero ticket number.
<code>requestID</code>	<code>int</code>	Unique identifier for this request (must be > 0). Used to track execution status via <code>GetRequestStatus()</code> .

Returns: `bool` - `true` if the close request was successfully queued, `false` if:

- The symbol is not registered
- The ticket is invalid or ≤ 0
- The queue is full (maximum 512 pending requests per symbol)

Error Handling: The request is queued immediately and validation (e.g., checking if the ticket is an open position) occurs during processing in the `Process()` method. If the ticket does not exist or is not an open position, the request ID is added to the fatal error buffer and `GetRequestStatus()` will return `REQ_STATUS_ERROR`.

Usage

```
// Close a position identified by its ticket number
ulong positionTicket = 123456;
bool queued = orderMgr.ClosePosition("EURUSD", positionTicket, 2001);
if(queued) Print("Close request queued for ticket ", positionTicket);

// Close multiple positions
orderMgr.ClosePosition("EURUSD", 123456, 2001);
orderMgr.ClosePosition("GBPUSD", 123457, 2002);
orderMgr.ClosePosition("AUDUSD", 123458, 2003);
```

DeletePendingOrder()

Signature:

```
bool DeletePendingOrder(string symbol, ulong ticket, int requestID)
```

Description: Submits a request to delete (cancel) a pending order. The delete request is queued and executed asynchronously. Before execution, the order manager verifies that the ticket exists and points to a pending

order. Use `GetRequestStatus()` with the `requestID` to check if the deletion was successful or is still pending.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The symbol of the pending order to delete. Must be a registered symbol.
<code>ticket</code>	<code>ulong</code>	The ticket/order ID of the pending order to delete. Must be a valid, non-zero ticket number.
<code>requestID</code>	<code>int</code>	Unique identifier for this request (must be > 0). Used to track execution status via <code>GetRequestStatus()</code> .

Returns: `bool` - `true` if the delete request was successfully queued, `false` if:

- The symbol is not registered
- The ticket is invalid or ≤ 0
- The queue is full (maximum 512 pending requests per symbol)

Error Handling: The request is queued immediately and validation (e.g., checking if the ticket is a pending order) occurs during processing in the `Process()` method. If the ticket does not exist or is not a pending order, the request ID is added to the fatal error buffer and `GetRequestStatus()` will return `REQ_STATUS_ERROR`.

Usage

```
// Delete a pending buy-limit order
ulong pendingOrderTicket = 123457;
bool queued = orderMgr.DeletePendingOrder("EURUSD", pendingOrderTicket, 2101);
if(queued) Print("Delete request queued for ticket ", pendingOrderTicket);

// Delete a buy-stop order
orderMgr.DeletePendingOrder("GBPUSD", 123458, 2102);
```

ModifyPosition()

Signature:

```
bool ModifyPosition(string symbol, ulong ticket, double sl, double tp, int
requestID)
```

Description: Submits a request to modify the stop loss and/or take profit levels of an open position. The modify request is queued and executed asynchronously. Both stop loss and take profit can be modified independently; set either to 0 to leave it unchanged. All prices are normalized to symbol specifications. Use `GetRequestStatus()` with the `requestID` to check if the modification was successful or is still pending.

Arguments:

Parameter	Type	Description
<code>symbol</code>	<code>string</code>	The symbol of the position to modify. Must be a registered symbol.
<code>ticket</code>	<code>ulong</code>	The ticket/order ID of the open position to modify. Must be a valid, non-zero ticket number.
<code>sl</code>	<code>double</code>	New stop loss price in quote currency. Set to 0 to leave the stop loss unchanged. If changed, must respect the symbol's minimum stop level distance.
<code>tp</code>	<code>double</code>	New take profit price in quote currency. Set to 0 to leave the take profit unchanged.
<code>requestID</code>	<code>int</code>	Unique identifier for this request (must be > 0). Used to track execution status via <code>GetRequestStatus()</code> .

Returns: `bool` - `true` if the modify request was successfully queued, `false` if:

- The symbol is not registered
- The ticket is invalid or ≤ 0
- The queue is full (maximum 512 pending requests per symbol)

Error Handling: The request is queued immediately and validation (e.g., checking if the ticket is an open position) occurs during processing in the `Process()` method. If the ticket does not exist, is not an open position, or the new SL/TP values violate stop level constraints, the request ID is added to the fatal error buffer and `GetRequestStatus()` will return `REQ_STATUS_ERROR`.

Usage

```
// Modify both SL and TP
ulong positionTicket = 123456;
bool queued = orderMgr.ModifyPosition("EURUSD", positionTicket, 1.0700, 1.0950, 2201);
if(queued) Print("Modify request queued");

// Modify only the stop loss (set TP to 0 to leave unchanged)
orderMgr.ModifyPosition("EURUSD", positionTicket, 1.0750, 0, 2202);

// Modify only the take profit (set SL to 0 to leave unchanged)
orderMgr.ModifyPosition("EURUSD", positionTicket, 0, 1.0900, 2203);

// Trail stop loss by moving it up as price moves
double currentPrice = SymbolInfoDouble("EURUSD", SYMBOL_BID);
double newSL = currentPrice - 0.0050;
orderMgr.ModifyPosition("EURUSD", positionTicket, newSL, 0, 2204);
```

GetRequestStatus()

Signature:

```
RequestStatus GetRequestStatus(int id)
```

Description: Queries the current status of a previously submitted request using its request ID. Checks the fatal error buffer first (O(1) lookup), then searches all trade and ticket queues across all symbols. Returns SUCCESS if the request ID is not found in any queue or error buffer, indicating the request has been fully processed. In live trading environments with proper error handling, SUCCESS usually indicates successful execution, though it may rarely indicate a never-queued request.

Arguments:

Parameter	Type	Description
<code>id</code>	<code>int</code>	The request ID to check. Must be the same ID used when submitting the original Trade(), ClosePosition(), DeletePendingOrder(), or ModifyPosition() call.

Returns: `RequestStatus` - One of three values: `REQ_STATUS_SUCCESS` (not in any queue or error buffer), `REQ_STATUS_PENDING` (found in a trade or ticket queue), or `REQ_STATUS_ERROR` (found in the fatal error buffer).

Usage

```
// Submit a trade and check its status
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0.1, 0, 1.0700, 1.0900, "Buy", 5001);

// Immediately check status (likely PENDING)
RequestStatus status = orderMgr.GetRequestStatus(5001);
if(status == REQ_STATUS_PENDING) Print("Request 5001 is still processing");

// Later in the next tick, check again
status = orderMgr.GetRequestStatus(5001);
if(status == REQ_STATUS_SUCCESS) Print("Request 5001 completed successfully");
else if(status == REQ_STATUS_ERROR) Print("Request 5001 encountered a fatal error");

// Check status of a close position request
orderMgr.ClosePosition("EURUSD", 123456, 5002);
status = orderMgr.GetRequestStatus(5002);
if(status == REQ_STATUS_ERROR) Print("Failed to close position 123456");
```

IsBusy()**Signature:**

```
bool IsBusy()
```

Description: Checks whether the order manager has any pending requests in its queues. Returns true if there are any unprocessed trade requests or ticket operations across any registered symbol. Useful for determining if the manager is idle and safe to perform other operations, or for implementing completion detection loops in expert advisors.

Arguments: None

Returns: `bool` - `true` if any trade or ticket queues contain pending requests, `false` if all queues are empty.

Usage

```
// Submit a trade request
orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0.1, 0, 1.0700, 1.0900, "Buy", 6001);

// Check if manager is busy
if(orderMgr.IsBusy()) {
    Print("Order manager still has pending requests");
} else {
    Print("Order manager is idle");
}

// Wait for all requests to complete
while(orderMgr.IsBusy()) {
    Print("Waiting for pending orders...");
    Sleep(100);
}
Print("All orders processed");
```

Process()

Signature:

```
void Process()
```

Description: Main processing loop that executes all queued trade requests and ticket operations. Must be called every tick from the OnTick() function of an expert advisor. This method processes ticket requests (closes, deletes, modifies) and trade requests (opens) for all registered symbols in sequence. In live trading, processing respects the configured time budget (default 200ms per tick) to avoid blocking other operations. In backtesting, timing constraints are removed for maximum throughput. Symbol information is refreshed automatically based on the environment (every new bar in tester, every 10 seconds in live trading).

Arguments: None

Returns: `void` - No return value.

Usage

```
// Must be called from OnTick() every tick
void OnTick() {
    // Process all queued trade requests and ticket operations
    orderMgr.Process();

    // Rest of EA logic follows...
    // Submit new trades, check conditions, etc.
}

// Example: Full tick cycle
void OnTick() {
    // 1. Process all pending orders
    orderMgr.Process();

    // 2. Check signal and submit new trade if needed
    if(BullishSignal()) {
        double ask = SymbolInfoDouble("EURUSD", SYMBOL_ASK);
        orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0, 0,
            ask - 50*_Point, ask + 100*_Point, "Buy Signal", nextID++);
    }

    // 3. Check existing positions and modify if needed
    if(ShouldTrailStop()) {
        // Modify trailing stop logic...
        orderMgr.ModifyPosition("EURUSD", positionTicket, newSL, 0, nextID++);
    }
}
```

Usage Example

```
#include <OrderManagerUpdate.mqh>

COrderManager orderMgr;

void OnInit() {
    // Initialize the order manager with EURUSD
    orderMgr.Init("EURUSD", 12345, 5, 50, 10, 200);

    // Add additional symbols
    orderMgr.AddSymbol("GBPUSD");
    orderMgr.AddSymbol("AUDUSD");

    // Configure risk management: 2% of balance per trade
    // When submitting trades with vol=0, volume will be auto-calculated
    orderMgr.SetRiskSettings(RiskPercentBalance, 2.0);
}
```

```
}

void OnTick() {
    // Process all queued requests
    orderMgr.Process();

    // Submit a market buy order with automatic volume calculation
    // Entry at market price, SL at Ask - 50 pips, TP at Ask + 100 pips
    // Since vol=0 and RiskPercentBalance is configured, volume is auto-calculated
    double curAsk = SymbolInfoDouble("EURUSD", SYMBOL_ASK);
    orderMgr.Trade("EURUSD", ORDER_TYPE_BUY, 0, 0,
        curAsk - 50 * _Point, curAsk + 100 * _Point,
        "Auto Volume Trade", 1001);

    // Check status of request ID 1001
    if(orderMgr.GetRequestStatus(1001) == REQ_STATUS_SUCCESS) {
        Print("Order 1001 executed successfully");
    }
    else if(orderMgr.GetRequestStatus(1001) == REQ_STATUS_PENDING) {
        Print("Order 1001 is still processing");
    }
}
```