



MQL for Traders

**Learn To Build a MetaTrader Expert Advisor in
One Day**

David M. Williams

A 3D geometric graphic consisting of several interconnected planes in shades of blue, orange, and grey, creating a complex, angular shape. The planes are arranged in a way that suggests depth and perspective, with some planes being more prominent than others. The overall effect is a modern, architectural design.

www.iExpertAdvisor.com

Table of Contents

Preface	6
About This Book	7
Standard Disclaimer	8
The MetaTrader Platform	9
Installation	9
Platform Applications.....	10
The Terminal	10
Client Server.....	12
The MetaEditor	13
Meta Query Language (MQL).....	16
MQL Basics	17
Data Types.....	17
Operator Reserved Words	19
The <i>if</i> and <i>else</i> Operators	19
The <i>switch</i> Operator	22
The <i>for</i> Loop Operator	22
The <i>while</i> Loop Operator	22
The <i>return</i> Operator.....	23
Functions.....	23
MQL Functions	26
Account Functions.....	26
Array Functions	28
Checkup Functions	28
Common Functions	28
Date and Time Functions	28
File Functions	30
Global Variable Functions	30
Math Functions	30
Object Functions	32
String Functions	32

Technical Indicators	32
Trading Functions.....	33
Window Functions	33
MQL Function Category Review.....	35
MQL Predefined Variables	36
MQL Constant Definitions.....	37
MQL Naming Conventions	37
iExpertAdvisor Naming Conventions	38
An Empty Expert Advisor	39
Expert Advisor Extern Variables.....	40
The iExpertAdvisor Framework.....	42
General System Logic.....	42
Trade Signals Variables	44
Open Trade Information Variables	45
fnOpenOrderInfo Function: Using the MQL Function <i>OrderSelect</i>	47
Trade Information Variables	55
fnOpenBuyLogic Function: Using the MQL Function <i>iMA</i>	58
fnOpenSellLogic Function: Using the MQL Function <i>iLowest</i>	63
fnCloseBuyLogic Function	68
fnCloseSellLogic Function	69
fnOrder Function: Using the MQL Functions <i>OrderSend</i> and <i>OrderClose</i>	70
The MQL Function <i>OrderSend</i>	70
The MQL Function <i>OrderClose</i>	74
Building the <i>iStarter</i> Expert Advisor.....	81
Comments in an MQL File.....	81
Property Definitions.....	82
Including External Code	82
The MQL <i>#include</i> Directive.....	82
The MQL <i>#import</i> Directive.....	83
Declaration of Variables.....	84
The <i>init</i> Function	84

The <i>start</i> Function.....	84
The <i>deinit</i> Function	85
The User Defined Functions.....	85
Expert Advisor Files.....	85
MQL Syntax Errors, Debugging and Testing.....	85
The Complete <i>iStarter</i> Expert Advisor Code	87
<i>iStarter</i> MQL code (continued)	88
<i>iStarter</i> MQL code (continued)	89
<i>iStarter</i> MQL code (continued)	90
Compiling the <i>iStarter</i> Expert Advisor	91
Running the <i>iStarter</i> Expert Advisor	92
Slightly Advanced Topics.....	93
Trailing Stop	93
Modifying a Buy Order.....	94
Multiple Trades/Single Trade.....	97
No Trade Signal to Close Position	100
Controlling Trade Open.....	100
Changing the <i>iStarter</i> Expert Advisor.....	101
Appendix 1 - Glossary	104
Appendix 2 – Reserved Words.....	108
Appendix 3 – Color Constants.....	109
Appendix 4 – Operation Precedence	110
Appendix 5 – Arithmetic Operators.....	111
Appendix 6 – Error Codes	112
Appendix 7 – Configuring Email.....	115
Appendix 8 – Preferred Prices	117
Appendix 9 – MarketInfo Function	118
Appendix 10 – Pending Order Definitions	120
Appendix 11 - UninitializeReason	121
Appendix 12 – Using the Comment Field.....	122
Appendix 13 – Using the <i>switch</i> Operator	123

Appendix 14 – Crossovers and Bouncing Values	124
Appendix 15 – Logging Information.....	127
Appendix 16 – The <i>iCustom</i> MQL Function.....	129

Preface

Writing the computer code to create an Expert Advisor has some similarities to writing a grammar school essay. They both have a very specific goal. For the Expert Advisor, the goal is to automate a trading system. For the essay, the goal is to communicate some information.

Each of these goals can be reached in many ways.

But when we are young - learning to write an essay for the first time - we are given a well-defined structure to follow. We are strongly advised to follow the rules of grammar until we are able to understand the subtle consequences of bending or breaking the rules. (My native language is English, but I assume that grammatical rules and structure of English are similarly applied to other languages).

In other words, when you get older, and have been writing for some time, you may bend or break the rules of grammar - as long as you fully understand the effect these changes will have on the *goal* of the essay.

After all, the real goal of the Essay is to *communicate* - not to write perfect grammar.

Similarly, the real goal of an Expert Advisor is automate a trading system - not to write perfect computer code.

The reason I make this comparison is because I am going to propose a framework for building an Expert Advisor. My own set of rules.

This framework is not the *only* way to build an Expert Advisor, as there are an infinite number of ways to build the same Expert Advisor. However, this framework and its rules provide a sound structure for a beginner. And like the student learning to write an essay, the trader learning to write an Expert Advisor should initially accept the rules as they are proposed. Later, as the trader's knowledge of MQL programming increases, she can challenge the rules and the structure and apply her own creativity to the task of building an automated trading system.

About This Book

This book makes no assumptions about the programming experience of the reader. The goal of this book is to teach a trader, with *no* programming experience, how to build a moderately sophisticated Expert Advisor in a single day.

This book is broken into modules, where each module addresses a specific goal. Generally each module builds upon information presented in previous modules.

At the start of each module, the objective of the module is defined along with a list of what will be covered. This is done so that a module can be skipped if the reader is familiar with the objectives. Similarly, a module can be reread if the needed.

In order to meet the requirement of teaching a trader to build an Expert Advisor in a single day many subjects are quickly reviewed. This does not mean the subject is not important - but it is not so important that a complete and thorough understanding can't be delayed until our first objective is met.

Within the module sections of the book, I have purposely tried to leave out as much information as possible. You don't hear *that* every day! In other words, the module sections of the book introduce only the information that is *essential* to learn how to build an Expert Advisor.

The appendix of the book is used to provide deeper information on selected topics along with expanded examples. Where appropriate, there are references to the appendix throughout the module sections of the book.

If your goal is to reach the point where you can build an Expert Advisor as fast as possible, then ignore all references to the appendix or advanced topics.

If you have more time and are interested in increasing your depth of MQL programming, then spending the time to review the advanced information in the appendix will be time well spent.

Note: The appendix of this book is a living document. Updates will be made to the appendix based on user feedback and the introduction of new programming techniques.

If you purchased this book directly from iExpertAdvisor, updates will be provided free of charge. If you "borrowed" this book from a friend and are still interested in receiving the updates, it is never too late to visit our website and purchase the book. We won't be mad.

I have gone through great effort to not use words or phrases specific to programming or MQL without defining them first. If you come across a word or phrase that you do not understand, and it is not defined in the book, it is probably not specific to programming or MQL. If you cannot find the meaning of a word, or you think it is a missed definition, please email me at: david.williams@iExpertAdvisor.com.

Standard Disclaimer

Please read and understand the following disclaimer:

This book should not be construed as personalized financial advice. Each trader must decide if the high level of risk associated with trading foreign currencies is appropriate for their financial situation.

iExpertAdvisor is not a registered investment advisor or broker/dealer. Readers are advised that the material contained herein should be used solely for informational purposes. iExpertAdvisor does not purport to tell or suggest which investment securities members or readers should buy or sell for themselves. Users should always conduct their own research and due diligence and obtain professional advice before making any investment decision. iExpertAdvisor will not be liable for any loss or damage caused by a reader's reliance on information obtained in any of our newsletters, special reports, email correspondence, or on our web site.

Our readers are solely responsible for their own investment decisions. The information contained herein does not constitute a representation by the publisher or a solicitation for the purchase or sale of securities. Our opinions and analyses are based on sources believed to be reliable and are written in good faith, but no representation or warranty, expressed or implied, is made as to their accuracy or completeness. All information contained in our newsletters or on our web site should be independently verified with the companies mentioned. The editor and publisher are not responsible for errors or omissions.

Past performance may not be indicative of future results. Therefore, you should not assume that the future performance of any specific investment or investment strategy will be profitable or equal to corresponding past performance levels.

The MetaTrader Platform

Module 1: The MetaTrader Platform

Objective: *Become familiar with the MetaTrader platform.*

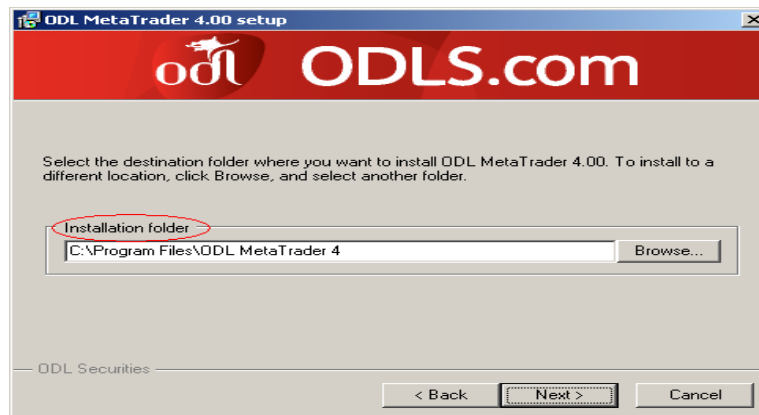
Subjects:

- Platform Installation
- Client-Server Definition
- The MetaEditor
- MQL
- Expert Advisor

Installation

The MetaTrader platform is available from many brokers. It is free to download and install. Some brokers will require an email address and then send a download link, while other brokers will immediately direct you to a download site. The best way to find a broker is to use a search engine and search for “MetaTrader Brokers”. Once you receive a download link to MetaTrader, installation is the same as any other Windows application.

For future reference, you may want to note *where* you install the MetaTrader platform. The following window is displayed during installation. This window allows the user to select the location of the MetaTrader platform. The location is referred to as the *MetaTrader Installation Directory* (or folder).



Platform Applications

The MetaTrader platform includes a trading platform (terminal.exe) and an editor named MetaEditor (metaeditor.exe). These applications can be launched from the Windows start menu, or the files can be double-clicked from the *MetaTrader Installation Directory*.



The Terminal

The MetaTrader trading platform is an intuitive, easy to use Forex trading platform. It offers what you would expect of a trading platform:

- The latest currency prices
- User configurable price charts
- Common technical indicators
- Order entry screens
- Trade history information
- Open trade status

The MetaTrader platform has four distinct sub windows. In addition to the main **Price Chart** window, the platform contains the **Market Watch** window, the **Navigator** window, and the **Terminal** window.

The **Market Watch** window displays the latest prices. Double clicking one of the symbols within the **Market Watch** window will open the **Order Entry** window.

The **Navigator** window allows access to **Accounts** (More than one account can be managed from a single platform), **Indicators**, **Expert Advisors**, **Custom Indicators** and **Scripts**.

Indicators, **Expert Advisors**, **Custom Indicators** and **Scripts** are attached to a **Price Chart** by double clicking the menu item.

The **Terminal** window displays up to seven tabs when an account is active. The tabs are **Trade**, **Account History**, **News**, **Alerts**, **Mailbox**, **Experts** and **Journal**.

The **Trade** tab is used to view the status of current open trades or pending trades. (Note: Right clicking the mouse within the **Trade** tab displays more viewing options. One useful option is the **Comment** field. This allows the user to view any comments inserted into a trade by an Expert Advisor. It can be useful to

iExpert Note

More Symbols may be displayed in the **Market Watch** window by right-clicking the mouse and selecting "Show All".

place the name of the Expert Advisor into the **Comment** field to help identify how a current trade was opened.)

The **Account History** tab is used to view closed trades. Again, the Comment field can be viewed by right clicking the mouse.

The **News** tab is used to display news sent to the platform from the broker.

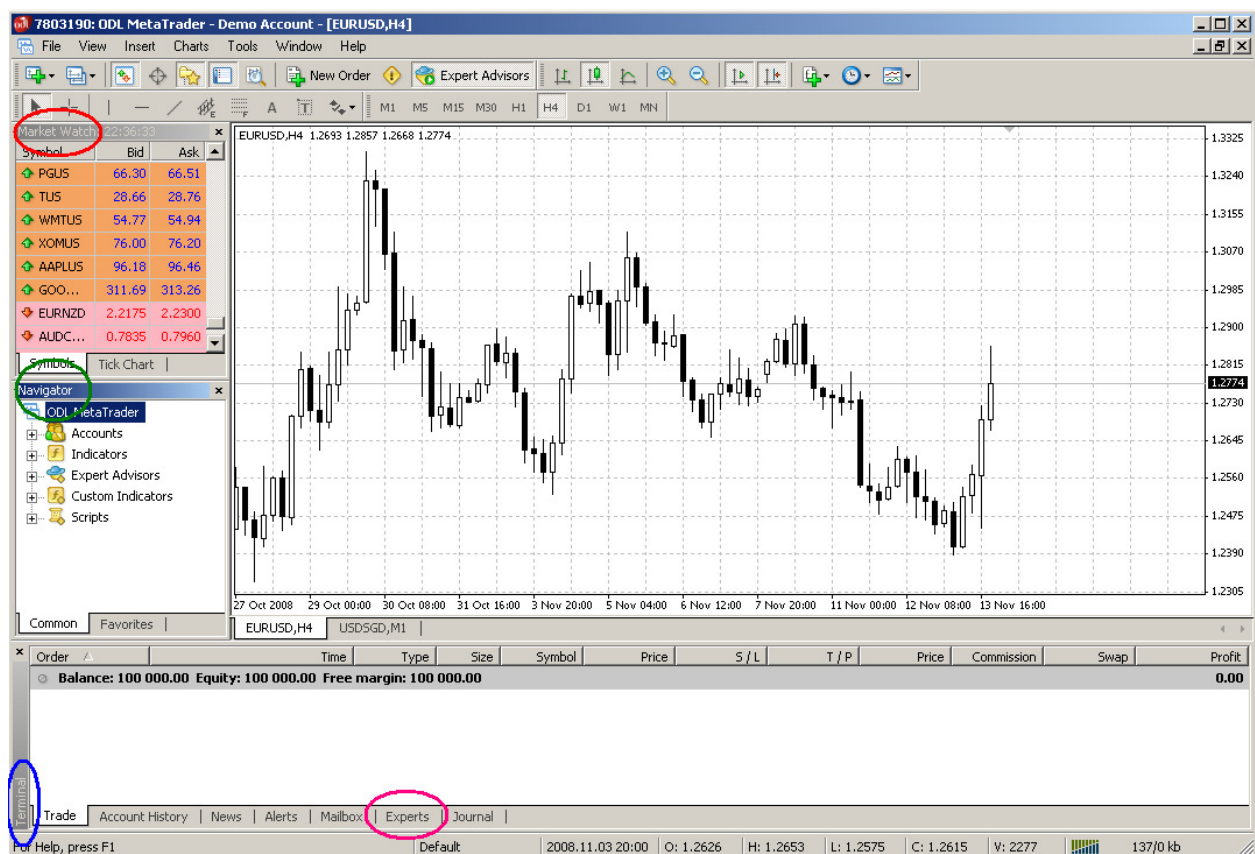
The **Alerts** tab is used to display Alerts that can be generated from Expert Advisors, Scripts and Indicators.

The **Mailbox** tab is used to display messages sent to the platform from the broker.

The **Experts** tab is used to display information from running Expert Advisors. This tab is very useful when developing and testing an Expert Advisor.

The **Journal** tab is used to display information relevant to the account such as logging in and out, and opening, modifying and closing trades.

This is a screen shot of the Terminal application.



Client Server

The MetaTrader terminal platform (terminal.exe) is a client of the MetaTrader broker's server.

The phrase client-server is used to describe the relationship between two (or more) processes running on a computer. Generally, a process is a running application, and usually these two processes are on two different computers.

The server process *serves* data, or provides a *service*, to clients. Usually a single server supports multiple clients.

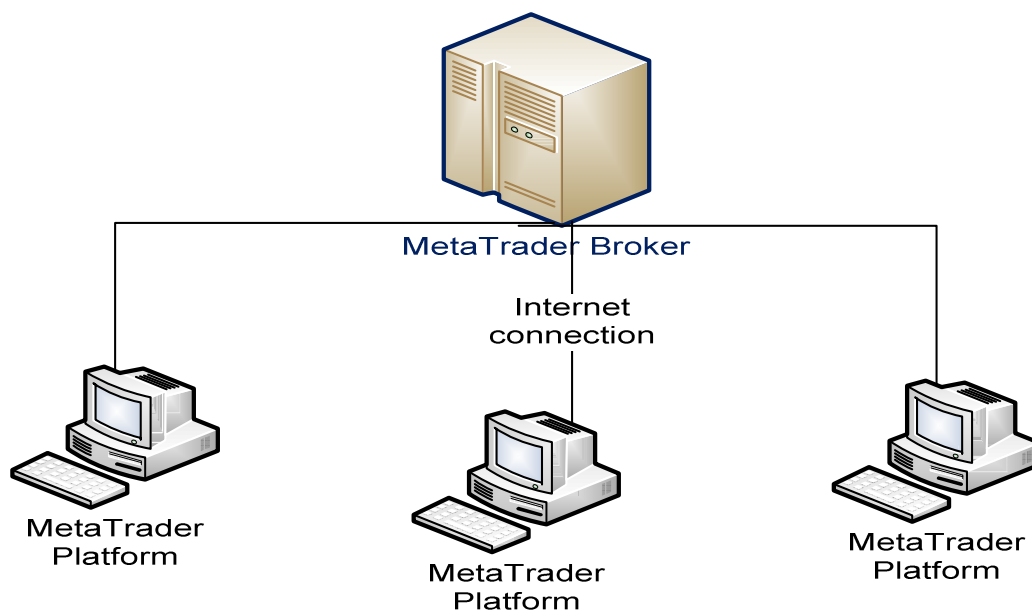
When the MetaTrader platform (the client) is started, it communicates with the broker's server. It attempts to login using a username and password. If the login is successful, the client and the server start communicating.

The server sends price information whenever there is updated price data. Each instance of new price information is called a **tick**. Eventually, when one full minute of ticks have arrived, a one minute **bar**, or **candle**, can be formed, showing the open, high, low and close prices.

The client sends requests to the server for account information as well as requests to open, modify and close trades.

iExpert Note

An Expert Advisor is invoked, or executed, on each incoming **tick**. If no ticks arrive, the Expert Advisor will not run. In slow markets, an Expert Advisor may be idle for long periods of time.



The MetaEditor

The MetaEditor is the application used to build Expert Advisors, Scripts and Custom Indicators. It is both a **text editor** and a **compiler**.

The **text editor** is useful for writing MQL code because it displays variables and functions in specific colors. Also, when entering an MQL function, the MetaEditor displays the parameters of the function. (Functions and parameters are covered in Module 2).

A **compiler** is a program that translates the human readable text of a programming language into a file that can be read and executed by a computer.

The human readable text must follow the compiler's rules of syntax.

The syntax specifies the human readable text of the programming language that is understood by the compiler. The syntax also specifies the special meaning of characters such as parenthesis, semi colons, etc.

The MetaTrader programming language is called Meta Query Language, or MQL. The MQL language is a collection of human readable keywords. (The syntax of MQL is similar to the syntax of the **C** programming language. Of course, if you are not familiar with the syntax of **C** this comparison is useless. However, after reading this book, you will have a very good understanding of MQL, and thereby, will be able to understand some of the syntax of **C**).

The MQL text files (sometimes called source code files) have a file extension of **mq4**. An Expert Advisor has a file extension of **ex4**.

The MetaEditor is used to create and edit **mq4** files. The MetaEditor has a button for compiling: when the MetaEditor compiles an **mq4** file it translates the text commands of the **mq4** file into an executable Expert Advisor's **ex4** file. (Both of these files are saved in the **experts** folder beneath the *MetaTrader Installation Directory*.)

The Expert Advisor **ex4** file is not exactly an executable file. It cannot be executed within Microsoft Windows. Rather, it is executed through the MetaTrader terminal platform.

In addition to the file editing window, the MetaEditor contains a **Navigator** and a **Toolbox** window.

The **Navigator** window provides MQL help via three tabs: **Files**, **Dictionary** and **Search**. The help is very good. All MQL functions are fully defined and there are very few errors in the documentation.

The Toolbox contains four tabs: **Errors**, **Find in Files**, **Online Library** and **Help**.

The **Errors** tab is an important window: when compiling, the MetaEditor writes messages to the **Error** tab of the **Toolbox**. Any syntax errors found during compilation are listed in this window. Double clicking the error message will focus the cursor on the offending line of MQL. Warning messages are also listed in the **Errors** window.

If there are any syntax errors present, the Expert Advisor's **ex4** file will not be built (created). However, warning messages alone will not prevent an Expert Advisor from building.

The focus of this book is learning MQL to build Expert Advisors. MQL is also used to build **Scripts**, **Templates**, **MQH** files, **Libraries** and **Custom Indicators**.

A **Script** is very similar to an Expert Advisor, except instead of running on each tick, it is run just once when it is selected.

A **Template** is a skeleton of MQL code designed to be the starting point for developing an MQL file. It usually contains the portions of the MQL file that are the same regardless of the specific purpose of the file.

An **MQH** file usually contains variable definitions, but no MQL code. (Variable definitions are introduced in Module 2.)

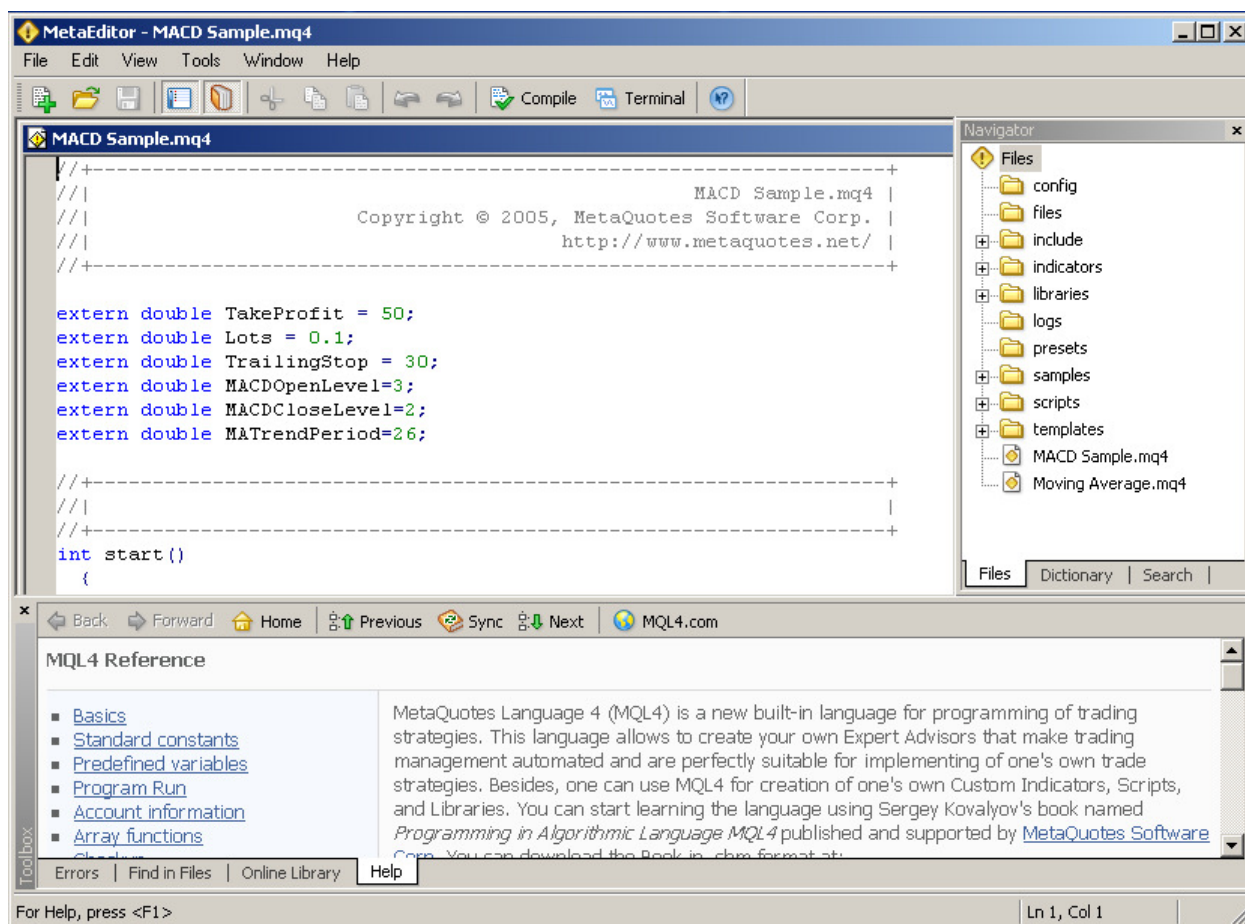
A **Library** is compiled MQL code (ex4) without an entry point. (Program entry points are introduced in Module 4.)

A **Custom Indicator** is similar to an Expert Advisor. Its MQL file has an **mq4** extension. It is built into an **ex4** file. It executes on each incoming tick. There are specific MQL constructs used for creating an indicator.

Selecting *File->New* from the MetaEditor will display the *Expert Advisor Wizard*. This wizard will guide a user to define input variables and then create an Expert Advisor shell. The user must then implement the complete logic of the EA using MQL.



This is a screen shot of the MetaEditor application.



Meta Query Language (MQL)

The MQL language is a programming language designed specifically for creating Expert Advisors and Indicators. The MQL language is well defined within the help resources of the MetaEditor. The language of MQL is similar to the C programming language.

MQL is considered a structured programming language. For the purpose of this book, the definition of a structure programming language is a language that supports *data types* and *functions*.

The remainder of this book focuses on the usage of the MQL language.

MQL Basics

Module 2: MQL Basics

Objective: *Understand the fundamentals of the MQL structured programming language.*

Subjects:

- **Data Types**
- **Operators (if/else for/while)**
- **User Defined Functions**

Most structured programming languages, including MQL, have a number of reserved words.

(The entire table of MQL Reserved Words can be seen in Appendix 2.)

A Reserved Word is a word that cannot be used within an Expert Advisor except for the specific purpose for which it was defined

The reserved words of MQL belong to one of three categories: Data Types, Operators and Memory Classes.

Data Types

MQL, like all programming languages, uses data types to store information. A data type is simply a section of memory used to store a value.

The MQL data types are:

Integer (*int*): An integer is a whole number. A whole number means the number does not have a decimal point. For example, the number of total open trades is an integer: 0, 1, 2 There is no concept of 1.5 open trades. However, the balance of a MetaTrader account is a number with a decimal point - such as 10,000.99. A number with a decimal point is called a *real* number in the field of mathematics. A *real* number data type is called a *double* in MQL.

Double (*double*): A double is a *real* number. A *real* number means the number has a decimal point. The balance of a MetaTrader account is a *real* number - such as 10,000.99.

Boolean (*bool*): A boolean is a data type that can one of two values: *true* or *false*.

String (*string*): A string is one or more characters. A string value is set using double quotes. For example: `message="Hello World"`.

Void (*void*): A void type is a “nothing” type. It is used in to indicate that no data type is inferred.

Date and Time (*datetime*): A datetime data type is used to store calendar dates and times. The data type itself is actually an integer. The current time is an integer whose value is the number of seconds elapsed since midnight January 1, 1970. (This is a common approach, used in many computer languages, to define the current time.)

Color (*color*): A color data type is actually an integer type. This data type is defined to make color assignments easy. The defined color types can be seen in Appendix 3.

Arrays: An array is a special form of a data type. An array can be thought of as a list. It can be a list of any of the other data types. All arrays have an index, which is the location of a specific item within an array (or) list of items.

Example of a *string* array with eight elements:

Index	Value (<i>string</i> data type)
0	“zero, or first”
1	“one”
2	“two”
3	“three”
4	“four”
5	“five”
6	“six”
7	“seven, or last”

Note: The index of an array is always an integer data type and the index begins at 0 – not 1.

MQL offers many functions for working with arrays. These functions are beyond the scope of this book. However, the concept of arrays must be understood to build even a trivial Expert Advisor. MQL makes use of many built-in arrays. An MQL built-in array is referred to as a *Series*.

Operator Reserved Words

The Operator reserved words are used to direct the flow of execution of an MQL Expert Advisor. These reserved words are *almost* human readable - once you understand their purpose.

The first category of *Operator* reserved words can be referred to as *conditional* expressions.

A conditional expression controls the flow of execution based on a condition, namely if something is true or false.

The *if* and *else* Operators

The reserved words *if* and *else* are used to define conditional expressions.

The *if* command will test the expression within its parenthesis. The expression will either be true or false. If the expression evaluates to true, then the code following the *if* statement will execute.

The *else* command is optional and is always paired with a corresponding *if* command. If the expression of the *if* command evaluates to false, then the code following the *else* statement will execute.

iExpert Note

The omission of curly braces is a classic bug committed by nearly all computer programmers at some point. Without curly braces, only the single line following the *if* expression will execute.

Curly braces are used in MQL to define a section (or define a scope) within an MQL file. The curly left brace `{` is used to start a section and the curly right brace `}` is used to end a section. Curly braces are used in pairs: there must always be a left and right curly brace.

Pairs of curly braces may be nested:

```
{  
    {  
    }  
}
```

The *if* and *else* conditional expressions may be used with or without curly braces. If curly braces are not used, the only statement executed following an *if* or *else* statement, when appropriate, is the next single line of code.

MQL code without curly braces:

```
int i;  
  
if( (i>0) )  
    Print ("The statement will print when i is greater than zero.");  
    Print ("The statement will always print.");
```

MQL code with curly braces:

```
int i;

if( (i>0) )
{
    Print ("The statement will print when i is greater than zero.");
    Print ("The statement will print when i is greater than zero.");
}
```

The conditional expression makes use of *boolean operators* to determine if the expression evaluates to true or false. The *boolean operators* used in MQL are:

Symbol	Definition
==	Equal To
!=	Not Equal To
&&	And
	Or
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To

Examples of simple conditional expressions:

```
// Example 1
if( Ask == Bid )
{
    Print("The Ask price is EQUAL to the Bid price.");
}
else
{
    Print("The Ask price is NOT EQUAL to the Bid price.");
}

// Example 2
if( Ask > Bid )
{
    Print("The Ask price is GREATER THAN the Bid price.");
}
else
{
    Print("The Ask price is GREATER THAN the Bid price.");
}
```

Conditional expressions can be simple or compound. A simple conditional expression evaluates a single expression. A compound expression evaluates two or more expressions. The expressions are usually related by the **boolean** operators AND (&&) or OR (||).

Example of a compound conditional expression:

```
int i;  
int j;  
  
if( (i>0) && (j<0) )  
{  
    Print ("The integer i is GREATER THAN zero AND j is LESS THAN zero.");  
}
```

As you can see, with the introduction of compound operators, a single *if/then* clause can become very complicated and confusing. Even experienced programmers can commit logical errors because of the **else** clause of a compound **if** statement. It is best for beginner programmers to keep these expressions as simple as possible (don't try to be clever).

A common bug often created in structured programming languages is to mistakenly use the assignment equal operator (=) instead of the conditional equal operator (==). However, this is not a problem for MQL programmers as an assignment within a conditional expression will generate a syntax error when compiling the Expert Advisor.

The *switch* Operator

The *switch* operator is a conditional operator that can be used when an if/else operator becomes too complicated. The *switch* operator is used with the reserved words *case*, *break* and *default*. The syntax of the *switch* operator is not overly complex, but its usage is not required to build most Expert Advisors. Further information about the *switch* operator can be found in Appendix 13.

The *for* Loop Operator

Loops are a useful programming construct when the same or similar action needs to be repeated a number of times. MQL offer two operators to support looping, the *for* operator and the *while* operator.

The *for* operator uses three expressions and is designed to run a fixed number of times. The syntax of the *for* operator is:

```
for( expresion1; expression2; expression3 )
```

Where:

- expression 1 is a counter
- expression 2 is a boolean statement
- expression 3 is usually an incrementer or decrementer

The *for* loop will execute as long as expression 2 is true.

Example:

```
for(int i=0; i<TotalTrades; i++)  
{  
    Print("i=", i );  
}
```

- On the first execution of this loop, the value of the integer *i* will be equal to 0.
- On each execution of the *for* loop, the value of *i* will be incremented.
- The *for* loop will continue to execute until the expression *i<TotalTrades* is no longer true.

The *while* Loop Operator

The *while* operator uses a single expression. The expression is a boolean statement. The *while* loop will continue to execute as long as the boolean expression is true.

Example:

```
int i = 0;
while( i < 10 )
{
    i++;
}
```

This loop continues to execute as long as the expression ($i < 10$) is true. Note it is the responsibility of the programmer to increment the integer i somewhere within the contents of the loop. If this increment is omitted, the **while** loop will execute forever.

Similar to the **if** operator, the **while** operator can evaluate compound expressions.

```
int i = 0;
int j = 10;
while( (i < 10) && (j > 0) )
{
    i++;
    j--;
}
```

In this case, the **while** loop will continue to execute as long as the integer i is less than ten **and** the integer j is greater than zero.

The return Operator

The **return** operator is used to return values from a function to the caller of the function. The data type of the return value is determined by the definition of the function's signature. The syntax is:

- **return**(0);
- **return** (true);
- **return**("hello world");

iExpert Note

Note: **while** loops are inherently more dangerous than **for** loops because with a slight omission they can run forever. Fortunately the MetaTrader platform checks for this condition (at run time) and will halt the execution of the Expert Advisor if this condition is detected.

The complete definition of a *function* and a *function signature* are covered in the following section.

Functions

MQL is considered a structured programming language. The MQL language supports *data types* and *functions*.

A function is a section of code. A function may require parameters and it can return a value.

As an example, a simple function could be created to add two numbers. The name of the function could be anything – but it is best to use a descriptive name that will help when reviewing the MQL code. In this case, a suitable name may be **Add**.

The *signature* of the **Add** function is:

```
int Add( int x, int y )
```

This function *signature* states that the **Add** function takes two *integer* parameters as input and *returns* an *integer*.

The MQL code of the **Add** function is:

```
int Add( int x, int y )
{
    int sum;
    sum = x + y;
    return(sum);
}
```

The **Add** function:

- declares an integer named **sum**
- adds the integer **x** and the integer **y** and stores the result in **sum**
- returns the integer **sum**

The **Add** function can be used like this:

```
int TotalTrades=0;
int NumberOfBuys=3;
int NumberOfSells=2;

TotalTrades = Add(NumberOfBuys, NumberOfSells);
```

This is a trivial example, but you can see the power of using functions to encapsulate program functionality. Notice how the user of the **Add** function has no knowledge of how the numbers are added – only that the two numbers passed into the function will be added and the sum is returned.

Functions are generally used to encapsulate (complex) program functionality, especially when the functionality is needed in more than one location.

An MQL Expert Advisor can be split up into a number of logical, special-purpose functions. Once a function has been created and tested, it can be used over and over again with little additional effort.

Functions that do not return a value are declared with the **void** type.

Example:

```
void PrintMessage()
```

The **PrintMessage** function accepts no parameters and does not return a value. The **return** reserved word operator may still be used within the function, but its usage is optional.

Example:

```
void PrintMessage()
{
    int i=0;

    if( i > 0 )
    {
        Print("Hello ", i );
        return;
    }
}
```

The framework presented in this book makes great use of user-developed functions, such as **Add** or **PrintMessage**, as well as the many functions provided by MQL language.

MQL Functions

Module 3: MQL Functions

Objective: *Become familiar with all MQL function categories.*

Subjects:

- **Introduction to MQL Functions**
- **MQL Predefined Variables**
- **MQL Constant Definitions**

MQL offers a large number of powerful built in functions. The folks at MetaTrader did a very good job of anticipating the needs of an Expert Advisor and providing the functions to meet those needs.

We will touch upon the MQL functions most often used in a typical Expert Advisor. Covering all MQL functions is beyond the scope of this book. (The *Navigator* window of the MetaEditor is an excellent resource for information and help for all of the MQL functions.)

However, there is a saying “you don’t know what you don’t know” – meaning if you don’t spend some time skimming through the MQL functions, you will not even be aware of what functionality is offered – or how the functionality can be used.

So without going too deep, here is an overview of the major MQL function categories.

Account Functions

- These functions are used to access to the active account information.
- These functions are used **often** in a typical Expert Advisor.

AccountBalance	AccountCredit	AccountCompany
AccountCurrency	AccountEquity	AccountFreeMargin
AccountFreeMarginCheck	AccountFreeMarginMode	AccountLeverage
AccountMargin	AccountName	AccountNumber
AccountProfit	AccountServer	AccountStopoutLevel
AccountStopoutMode		

Array Functions

- These functions are used to work with arrays.
- These functions are used **rarely** in a typical Expert Advisor.

ArrayBsearch	ArrayCopy	ArrayCopyRates
ArrayCopySeries	ArrayDimension	ArrayGetAsSeries
ArrayInitialize	ArrayIsSeries	ArrayMaximum
ArrayMinimum	ArrayRange	ArrayResize
ArraySetAsSeries	ArraySize	ArraySort

Checkup Functions

- These functions are used to determine the status of the client terminal.
- These functions are used **often** in a typical Expert Advisor.

GetLastError	IsConnected	IsDemo
IsDllsAllowed	IsExpertEnabled	IsLibrariesAllowed
IsOptimization	IsStopped	IsTesting
IsTradeAllowed	IsTradeContextBusy	IsVisualMode
UninitializeReason		

Common Functions

- These are general purpose functions that do not fall into another category.
- These functions are used **often** in a typical Expert Advisor.

Alert	Comment	GetTickCount
MarketInfo	MessageBox	PlaySound
Print	SendFTP	SendMail
Sleep		

Date and Time Functions

- These functions are used to work with the datetime data type.
- These functions are used **often** in a typical Expert Advisor.

Day	DayOfWeek	DayOfYear
------------	------------------	------------------

Hour	Minute	Month
Seconds	TimeCurrent	TimeDay
TimeDayOfWeek	TimeDayOfYear	TimeHour
TimeLocal	TimeMinute	TimeMonth
TimeSeconds	TimeYear	Year

File Functions

- These functions are used to work with files on the client's file system.
- These functions are used *rarely* in a typical Expert Advisor.

FileClose	FileDelete	FileFlush
FileIsEnding	FileIsLineEnding	FileOpen
FileOpenHistory	FileReadArray	FileReadDouble
FileReadInteger	FileReadNumber	FileReadString
FileSeek	FileSize	FileTell
FileWrite	FileWriteArray	FileWriteDouble
FileWriteInteger	FileWriteString	

Global Variable Functions

- These functions are used to work with MQL global variables.
- These functions are used *rarely* in a typical Expert Advisor.

GlobalVariableCheck	GlobalVariableDel	GlobalVariableGet
GlobalVariableName	GlobalVariableSet	GlobalVariableSetOnCondition
GlobalVariablesDeleteAll	GlobalVariablesTotal	

Math Functions

- These functions are used to calculate the value of common mathematic formulas.
- These functions are used *rarely* in a typical Expert Advisor.

MathAbs	MathArccos	MathArcsin
MathArctan	MathCeil	MathCos
MathExp	MathFloor	MathLog
MathMax	MathMin	MathMod
MathPow	MathRand	MathRound
MathSin	MathSqrt	MathSrand

Object Functions

- These functions are used to work with graphical objects on the current chart.
- These functions are used *rarely* in a typical Expert Advisor.

ObjectCreate	ObjectDelete	ObjectDescription
ObjectFind	ObjectGet	ObjectGetFiboDescription
ObjectGetShiftByValue	ObjectGetValueByShift	ObjectMove
ObjectName	ObjectsDeleteAll	ObjectSet
ObjectSetFiboDescription	ObjectSetText	ObjectsTotal
ObjectType		

String Functions

- These functions are used to work with the string data type.
- These functions are used *rarely* in a typical Expert Advisor.

StringConcatenate	StringFind	StringGetChar
StringLen	StringSetChar	StringSubstr
StringTrimLeft	StringTrimRight	

Technical Indicators

- These functions are used to calculate the value of common technical indicators.
- These functions are used *often* in a typical Expert Advisor.

iAC	iAD	iAlligator
iADX	iATR	iAO
iBearsPower	iBands	iBandsOnArray
iBullsPower	iCCI	iCCIOnArray
iCustom	iDeMarker	iEnvelopes
iEnvelopesOnArray	iForce	iFractals
iGator	ilchimoku	iBWMFI

iMomentum	iMomentumOnArray	iMFI
iMA	iMAOnArray	iOsMA
iMACD	iOBV	iSAR
iRSI	iRSIOnArray	iRVI
iStdDev	iStdDevOnArray	iStochastic
iWPR		

Trading Functions

- These functions are used to manage trades.
- These functions are used *often* in a typical Expert Advisor.
- These functions can be divided into two categories: *Trade Commands* and *Trade Information*.

Trade Command Functions

OrderClose	OrderCloseBy	OrderDelete
OrderModify	OrderPrint	OrdersTotal
OrderSelect	OrderSend	OrdersHistoryTotal

Trade Information Functions

OrderClosePrice	OrderCloseTime	OrderComment
OrderCommission	OrderExpiration	OrderLots
OrderMagicNumber	OrderOpenPrice	OrderOpenTime
OrderProfit	OrderStopLoss	OrderSwap
OrderSymbol	OrderTakeProfit	OrderTicket
OrderType		

Window Functions

- These functions are used to work with the current chart window.
- Aside from the *Period* and *Symbol* functions, these functions are used *rarely* in a typical Expert Advisor.

HideTestIndicators	Period	RefreshRates
--------------------	--------	--------------

Symbol	WindowBarsPerChart	WindowExpertName
WindowFind	WindowFirstVisibleBar	WindowHandle
WindowIsVisible	WindowOnDropped	WindowPriceMax
WindowPriceMin	WindowPriceOnDropped	WindowRedraw
WindowScreenShot	WindowTimeOnDropped	WindowsTotal
WindowXOnDropped	WindowYOnDropped	

MQL Function Category Review

This table summarizes the MQL function categories, their typical usage and their level of difficulty.

Function Category	Used in a Typical Expert Advisor	Level of Difficulty
Account	Sometimes	Easy
Array	Rarely	Hard
Checkup	Often	Easy
Common	Often	Medium
Date and Time	Sometime	Medium
File	Rarely	Hard
Global	Rarely	Hard
Math	Rarely	Medium
Object	Rarely	Hard
String	Rarely	Medium
Indicators	Often	Hard
Trading	Often	Hard
Windows	Rarely	Hard

In reviewing this table you may have noticed two categories which are used *often* and are *hard* to work with: the *Trading Functions* and the *Indicator Functions*. For this reason, special attention is given to these function categories.

MQL Predefined Variables

MQL defines a number of variables that are specific to a trading platform. These variables make it easy to access typical trading platform data – needed by almost all Expert Advisors.

Note: All of these variables contain the value for the currency pair of the price chart to which the Expert Advisor is attached.

Variable Name	Data Type	Description
Ask	double	The last Ask price.
Bars	integer	The number of the currently forming Bar on the chart.
Bid	double	The last Bid price.
Close	array of doubles	The close price of each bar on the current chart.
Digits	integer	Number of digits after decimal point for the current symbol price.
High	array of doubles	The high price of each bar on the current chart.
Low	array of doubles	The low price of each bar on the current chart.
Open	array of doubles	The open price of each bar on the current chart.
Point	double	The current symbol point value in the quote currency
Time	array of datetimes	The open time of each bar on the current chart.
Volume	Array of integers	The tick volume of each bar on the current chart.

MQL Constant Definitions

MQL defines a number of predefined constants that are used by most Expert Advisors.

Constant Name	Data Type	Value	Description
OP_BUY	Integer	0	Identifier for a buy position.
OP_SELL	Integer	1	Identifier for a sell position.
OP_BUYLIMIT	Integer	2	Identifier for a buy limit pending order.
OP_SELLLIMIT	Integer	3	Identifier for a sell limit pending order.
OP_BUYSTOP	Integer	4	Identifier for a buy stop pending order.
OP_SELLSTOP	Integer	5	Identifier for a sell stop pending order.

These constant definitions are used with MQL functions such as **OrderSend** and **OrderType**.

MQL Naming Conventions

You may have noticed that some of the MQL functions follow a naming convention. For example, all of the technical indicator functions start with a lower case *i*, for example **IMA()**, **iADX**, etc.

Following a strict naming convention can increase the readability of your MQL code. Readability is important because the easier you can read and understand the MQL code the faster you can make changes and find bugs. Actually, as most experienced programmers are aware, a typical programmer spends about 10% of their time writing a program and 90% of their time debugging! For this reason alone, it is worth taking the time to follow a naming convention to ease debugging.

A naming convention can be applied to both user-defined functions and variables. Generally, a variable or function name can have *any* name as long as it is not a reserved word or a pre-defined variable. Also, a function or variable name cannot start with a number, a special character, or one of the MQL defined operator characters.

iExpertAdvisor Naming Conventions

The iExpertAdvisor naming convention for user-defined functions stipulates that the name start with the letters **fn**. The remainder of the function name follows the *camel case* convention. For example, a function to get the highest price of a currency pair might be called **fnGetHighestPrice**.

The iExpertAdvisor naming convention for user-defined variable stipulates that the name start with the starting letter of the variable's data type. The remainder of the variable name follows the *camel case* convention.

- An **integer** data type variable starts with a lower case **i**. For example, **iNumberOfTrades**.
- A **double** data type variable starts with a lower case **d**. For example, **dLowestBalance**.
- A **string** data type variable starts with the lower case letters **str**. For example, **strMessage**.
- A **boolean** data type variable starts with a lower case **b**. For example: **bOkToTrade**.
- A **datetime** and **color** data type variables start with a lower case **i**. (These types are actually integer types.) For example **iCloseBuyColor**.
- An array starts with a lower case **a**. For example, **aTradeList**.

Using the wrong data type can cause a bug. For example, the value used to define a lot size should be a double data type. However the MQL function used to open an order will (silently) accept an integer data type for the lots parameter. This will work fine as long as the orders are opened with lot sizes of 1, 2, 3, etc. If an order is placed to open with a lot size of 1.5 lots, the MetaTrader platform will automatically convert the value from 1.5 to 1, and the order will be opened with the incorrect lot size of 1.0.

An Empty Expert Advisor

Module 4: An Empty Expert Advisor

Objective: *Learn the structure of a basic Expert Advisor.*

Subjects:

- *init, start and deinit Functions*
- *extern Variables*

If you use the MetaEditor's *Expert Advisor Wizard* to build an Expert Advisor, it will create an Expert Advisor with three empty functions. These special functions are:

- `init`
- `start`
- `deinit`

init is short for initialization. The ***init*** function is only called once - the first time the Expert Advisor is attached to a chart.

start is the entry point of the Expert Advisor. Each time the Expert Advisor is run, program execution begins at the ***start*** function.

deinit is short for de-initialization. The ***deinit*** function is called once when the Expert Advisor is removed from the chart, or for some other reason that causes the Expert Advisor to stop executing.

Note: An Expert Advisor ***start*** function is executed each time a new tick of data is received from the trade server. In very slow markets, it is not unusual for an Expert Advisor to not execute for several minutes due to a lack of incoming price data (ticks).

There are many reasons an Expert Advisor may stop executing. Each reason can be identified using the MQL function ***UninitializeReason***. This function is covered in greater detail in Appendix 11.

The ***init*** function is useful for setting variables that will be needed later by the Expert Advisor. One usage of the ***init*** function is to store the time, or the ***Bar***, when the Expert Advisor was first started.

None of these functions *need* to be within an Expert Advisor for the MQL code to compile. Depending on what other code is present in the Expert Advisor, you may or may not receive a warning message such as "*Start function not found and cannot be run*". Of course, if there is no ***start*** function, the Expert Advisor will not execute because there is no entry point.

Copyright© *iExpertAdvisor*, LLC All Rights Reserved.

Many Expert Advisor do not use the ***init*** or the ***deinit*** functions. These functions are not always needed to meet the requirements of the Expert Advisor's trading logic.

Expert Advisor Extern Variables

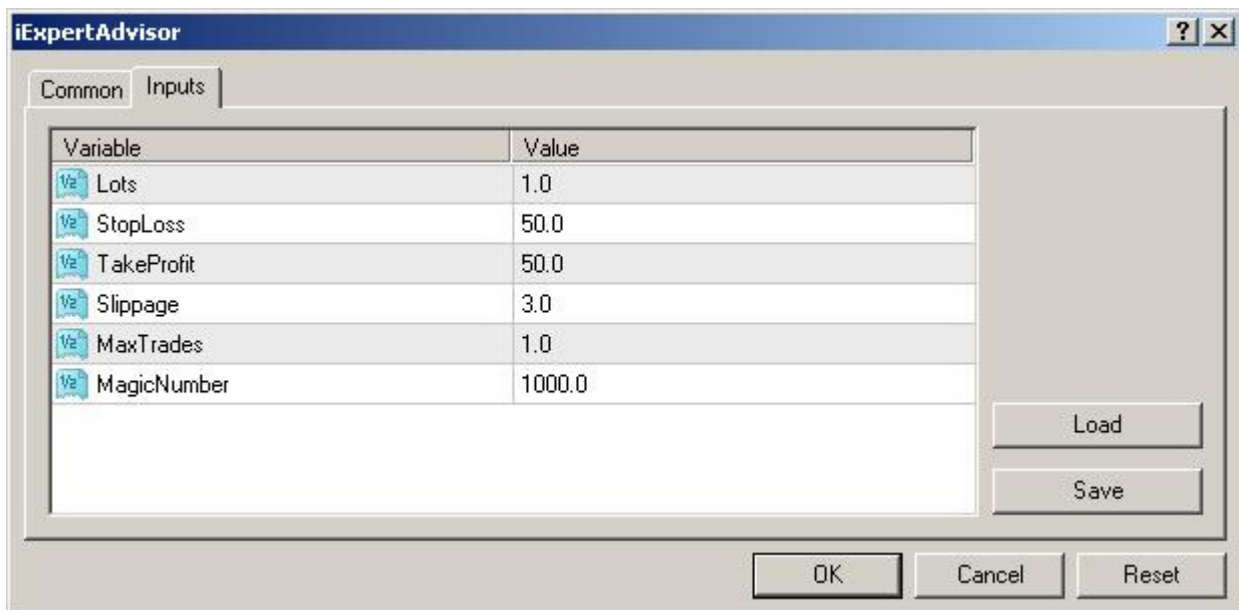
Variables that are defined as ***extern*** are the variables that appear as user input variables when the Expert Advisor is first attached to a chart. Any variable data type can be declared as an ***extern***. There are a number of user variables that are required of almost all Expert Advisors. For this reason, when starting a new Expert Advisor, I typically always add these variables to the start of the file.

Name	Data Type	Default Value	Description
dLots	double	1.0	The lot size to use when opening new positions.
iStopLoss	int	50	The default StopLoss of an open position.
iTakeProfit	int	50	The default TakePofit of an open position.
iSlippage	int	3	The maximum number of points the price can move when attempting to open position.
iMaxTrades	int	1	The maximum number of open positions allowed.
iMagicNumber	Int	1000	A unique number that allows the EA to identify the positions it has opened.

This is the MQL source code needed to define the variables.

```
// User Set-able Variables
extern double dLots=1.0;
extern double iStopLoss=50;
extern double iTakeProfit=50;
extern double iSlippage=3;
extern double iMaxTrades=1;
extern double iMagicNumber=1000;
```


This is the screen displayed when attaching the Expert Advisor to a chart. Making a variable available for user selection on this screen is as easy as inserting the reserved word **extern** in front of the variable declaration.



The screenshot shows the 'iExpertAdvisor' dialog box with the 'Inputs' tab selected. It contains a table with two columns: 'Variable' and 'Value'. The table lists six variables: Lots, StopLoss, TakeProfit, Slippage, MaxTrades, and MagicNumber, each with a corresponding value. To the right of the table are 'Load' and 'Save' buttons. At the bottom of the dialog are 'OK', 'Cancel', and 'Reset' buttons.

Variable	Value
Vz Lots	1.0
Vz StopLoss	50.0
Vz TakeProfit	50.0
Vz Slippage	3.0
Vz MaxTrades	1.0
Vz MagicNumber	1000.0

Note: This is a screen shot of a custom Expert Advisor developed for a client. When developing an Expert Advisor for an end user that has no knowledge of MQL (and has no desire to learn MQL) it is usually better to omit the variable type naming convention for the **extern** variables.

The iExpertAdvisor Framework

Module 5: The iExpertAdvisor Framework

Objective: *Learn the framework of the iExpertAdvisor method of building Expert Advisors.*

Subjects:

- **fnOpenOrderInfo**
- **fnOpenBuyLogic**
- **fnOpenSellLogic**
- **fnCloseSellLogic**
- **fnOrder**

The iExpertAdvisor framework used for building an Expert Advisor uses a number of narrowly defined functions to perform specific tasks and two groups of variables to store data and control the flow of execution.

Although the iExpertAdvisor functions may return a value, the method purposely does not control the flow of execution based on the function return values. This was done to simplify writing, maintaining and understanding the MQL code.

General System Logic

The general system logic of the iExpertAdvisor framework is straightforward. The Expert Advisor developed for this book is named **iStarter**. The following seven steps are followed in order .

1. Get any open trade information.

- The MetaTrader server is contacted to get the latest information about any open trades that the Expert Advisor is interested in managing.
- The information is stored in variables that can be accessed from anywhere within the Expert Advisor.
- If there are no open trades then no information is stored.

2. Check the criteria for opening a buy position.

- Using the latest tick data, any indicators or formulas are calculated to determine if a buy position should be opened.
- If the evaluation of the buy criteria indicates a position should be opened a value is stored in a variable that can be accessed from anywhere within the Expert Advisor.

Copyright© iExpertAdvisor, LLC All Rights Reserved.

3. Check the criteria for opening a sell position.

- Using the latest tick data, any indicators or formulas are calculated to determine if a sell position should be opened.
- If the evaluation of the sell criteria indicates a position should be opened a value is stored in a variable that can accessed from anywhere within the Expert Advisor.

4. Check the criteria for closing a buy position.

- Using the latest tick data, any indicators or formulas are calculated to determine if a buy position should be closed.
- If the evaluation of the buy criteria indicates a position should be closed, a value is stored in a variable that can accessed from anywhere within the Expert Advisor.

5. Check the criteria for closing a sell position.

- Using the latest tick data, any indicators or formulas are calculated to determine if a sell position should be closed.
- If the evaluation of the sell criteria indicates a position should be closed, a value is stored in a variable that can accessed from anywhere within the Expert Advisor.

6. Open or close new positions.

- If the criteria have been met to open or close a position, a position is opened or closed.

7. Manage the open trades.

- Apply trailing stops, etc. to open positions.

Trade Signals Variables

The variables used to control the Expert Advisor execution are based on the concept of a *trade signal*. An integer variable named *iTradeSignal* is defined. This variable is used to control most of the logic of the Expert Advisor.

To improve the readability of the code, the values that can be assigned to the *iTradeSignal* variable are defined as integer variables. These variables maintain a constant value, that is, their value is never changed. They are defined simply to make the MQL easier to understand.

The variables that can be assigned to the *iTradeSignal* variable are:

Variable Name	Constant Value
iOpenBuySignal	100
iCloseBuySignal	-200
iOpenSellSignal	300
iCloseSellSignal	-400
iNoSignal	0

Within the middle of several hundred lines of MQL code, consider trying to evaluate the correctness of these two statements:

- if(*iTradeSignal* == 300) ...
- if(*iTradeSignal* == *iCloseSellSignal*) ...

The second statement, using the *iCloseSellSignal* variable, requires much less effort to understand.

The MQL code used to define the trade signal variables:

```
// Trade Signal Variables
int iTradeSignal=0;
int iOpenBuySignal=100;
int iCloseBuySignal=-200;
int iOpenSellSignal=300;
int iCloseSellSignal=-400;
int iNoSignal=0;
```

Open Trade Information Variables

Most Expert Advisors are interested in managing open trades. In order to manage these trades, the Expert Advisor must have complete information about them.

This group of variables is defined to hold the current information about the open trades in which the Expert Advisor is interested. Variables can be added or subtracted from this category based on the specific needs of the Expert Advisor. The list is an attempt to include the most commonly needed information.

Variable Name	Variable Type	Description
iTotalTrades	int	Total number of trades opened for the account.
dTotalTradeLots	double	Total number of lots managed by the Expert Advisor.
dTotalTradeSwap	double	Total amount of swap of the open trades managed by the Expert Advisor.
iTotalBuyTrades	int	Total number of Buy positions managed by the Expert Advisor.
iTotalSellTrades	int	Total number of Sell positions managed by the Expert Advisor.
dLastTradeOpenPrice*	double	The open price of the last trade opened by the Expert Advisor.
iLastTradeOpenTime*	datetime	The open time of the last trade opened by the Expert Advisor.
dLastTradeStopLoss*	double	The stoploss of the last trade opened by the Expert Advisor.
dLastTradeTakeProfit*	double	The takeprofit of the last trade opened by the Expert Advisor.
dLastTradeType*	double	The trade type (Buy,Sell) of the last trade opened by the Expert Advisor.
dLastTradeTicket*	int	The ticket of the last trade opened by the Expert Advisor.
strLastTradeSymbol*	string	The Symbol of the last trade opened by the Expert Advisor.
strLastTradeComment*	string	The Comment of the last trade opened by the Expert Advisor.
dTotalOpenProfit	double	Total amount of profit of the open trades managed by the Expert Advisor.
iTotalSelectedTrades	Int	Total number of trades managed by the Expert Advisor.

* These variables are not required for most basic Expert Advisors.

Note: An Expert Advisor generally connects to the broker's server once during each invocation, so the information about the open trades collected at the start of execution is valid for the entire invocation. In other words, there is no value in connecting to the server more than once per invocation

These are the variable declarations of the Trade Information variables.

```
// Open Trade Information Variables  
  
int iTotalTrades;  
double dTotalTradeLots;  
double dTotalTradeSwap;  
int iTotalBuyTrades;  
int iTotalSellTrades;  
double dTotalOpenProfit;  
int iTotalSelectedTrades;
```

fnOpenOrderInfo Function: Using the MQL Function *OrderSelect*

Module 6: The *fnOpenOrderInfo* Function

Objective: *Learn to use the MQL functions *OrdersTotal* and *OrderSelect*.*

Subjects:

- *OrdersTotal*
- *OrderSelect*
- Order Ticket
- Using a *for* loop
- The Trade Information Variables

The function *fnOpenOrderInfo* returns no value and does not require any input parameters. The empty function looks like:

```
// Get information about all open trades  
  
void fnGetOpenTradeInfo()  
{  
  
}
```

The function *fnOpenOrderInfo* is used to contact the MetaTrader server and collect the latest information about any trades that the Expert Advisor is interested in managing.

An Expert Advisor can be configured to manage many sets of positions, including, but not limited to:

- all positions for an account
- all positions for a currency symbol
- all positions for a magic number
- all buy positions
- all sell positions

- the specific ticket number of a position
- positions opened during specific time frames
- any combination of the above criteria

The magic number is a unique number that is assigned to an order when it is opened. For example, when an Expert Advisor opens an order on the **EURUSD**, it can assign a magic number of 3000. Later when the Expert Advisor requests information about all of the open orders for the account, it recognizes the orders it has opened by their magic number - any order whose magic number is 3000. The magic number allows an Expert Advisor a method to tag and identify orders.

Notice that there is no easy method to manage trades based on the time frame of the chart on which they were opened. There is however an advanced method of defining very specific trade sets. This method has some caveats and is covered in Appendix 12.

Note: Each order has a unique ticket number that is generated by the MetaTrader server when the order is first created.

The most common trade set for an Expert Advisor to manage is the set all positions for a currency symbol, followed closely by all positions for a currency symbol and magic number.

The iExpertAdvisor method uses a set defined as all positions for a currency symbol and magic number.

The iExpertAdvisor Expert Advisor uses the MQL functions **OrdersTotal** and **OrderSelect** to collect information about the open positions on the account.

OrdersTotal is a simple MQL function. It requires no input parameters and returns an integer whose value is the total number of open positions for the account. (Note: this number includes pending orders.) The function signature is:

```
int OrdersTotal( )
```

Using the **iTotalTrades** variable declared earlier, the **OrdersTotal** function is used to set the **iTotalTrades** variable.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    // Get the total number of trades
    iTotalTrades = OrdersTotal();
}
```


The MQL function **OrderSelect** is not simple. **OrderSelect** returns a **boolean** value, requires two parameters and allows an optional third parameter. The function signature is:

```
bool OrderSelect( int index, int select, int pool=MODE_TRADES)
```

The **OrderSelect** function returns a boolean value of true or false. If the function returns false, it generally means the function failed and its results should not be used.

The first parameter is an integer and defines the index (into the order pool) of the order to select. The value of this number can be from 0 to **iTotalTrades**.

The second parameter is an integer and is the requested sort order of the list of **order pools**. The possible values are:

- **SELECT_BY_POS** – ordered by the way the orders were added to the pool.
- **SELECT_BY_TICKET** – ordered by the ticket number.

The **order pools** are arrays data types (or series) and can be visualized as lists.

This is an example of the **SELECT_BY_POS** order pool. The orders may be listed chronologically, but it is not guaranteed.

Index	Ticket Number	Open Time
0	900500	11940000
1	900400	11940100
2	900600	11940200
3	900300	11940300

This is an example of the **SELECT_BY_TICKET** order pool. The orders are listed by ticket number.

Index	Ticket Number	Open Time
0	900300	11940300
1	900400	11940100
2	900500	11940000
3	900600	11940200

The third parameter of the **OrderSelect** function is an integer and defines the pool of orders to select from.

The options are:

- **MODE_TRADES** (default)- order are selected from the trading pool (opened and pending orders),
- **MODE_HISTORY** – orders are selected from the history pool (closed and canceled orders). This mode is only used with the **SELECT_BY_POS** parameter.

The function syntax “`int pool=MODE_TRADES`” defines the default **pool** parameter as **MODE_TRADES**. This means if the parameter is not explicitly set, it will be set to **MODE_TRADES**.

In order to move through the entire list of the order pool (by index) a **for** loop is used.

- An **integer** value name **pos** is defined for the counter that is the index into the pool.
- The boolean test used to detect the end of the list used the variable **iTotalTrades**.

The syntax of the **for** loop is:

```
for(int pos=0;pos<iTotalTrades;pos++)
```

The **OrderSelect** function is placed inside of the loop to read each order in the order pool.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    iTotalTrades = OrdersTotal();

    for(int pos=0;pos<iTotalTrades;pos++)
    {
        OrderSelect(pos, SELECT_BY_POS);
    }
}
```

Since the **OrderSelect** function can fail, it is best to capture the return value and control the flow of execution based on the success of the **OrderSelect** function.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    iTotalTrades = OrdersTotal();

    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos, SELECT_BY_POS)==false)
            continue;
    }
}
```

This code tests the return value from the *OrderSelect* function. If the function returns *false*, that iteration of the *for* loop is skipped by using the *continue* reserved word.

(You may have noticed the integer parameter, *pos*, used to increment the *for* loop, does not follow the naming convention for an integer. Names of variables used as counters are often excluded from the naming convention.)

At this point we have used the *OrdersTotal* and *OrderSelect* functions to query the MetaTrader server about open positions on the account. Now, within the loop, if the order has been selected successfully, we can gather further information about the position. The section **Trade Information Functions** defines all of the order information that can be request after a successful call to *OrderSelect*.

The first task is to apply filtering to the trade set. That is, since the loop will be iterating through *all* orders for the account, there may be some orders in which the Expert Advisor is not interested. The *iStarter* Expert Advisor uses the currency symbol and the magic number to define the *Trade Set*. (The *Trade Set* is defined as the collection of open positions that the Expert Advisor manages).

- The string value of currency symbol of the chart to which the Expert Advisor is attached is available through the *Symbol* MQL function.
- The string value currency symbol of the currently selected order is available through the *OrderSymbol* MQL function.
- These values can be compared to determine if the Expert Advisor is interested in the order.
- Both of these functions require no input parameters and return a string.
- The following code, placed with the for loop checks if the chart symbol is not equal to the order symbol. If they are not equal, the iteration of the *for* loop is skipped.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    iTotalTrades = OrdersTotal();

    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos,SELECT_BY_POS)==false)
            continue;

        if(Symbol() == OrderSymbol())
        {
            // Save order information here
        }
    }
}
```

The second filtering criteria applied of the **iStarter** Expert Advisor is the magic number. The external integer variable **iMagicNumber**, set by the user when configuring the Expert Advisor, is compared to the magic number of the orders using the **OrderMagicNumber** MQL function.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    iTotalTrades = OrdersTotal();

    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos,SELECT_BY_POS)==false)
            continue;

        if( (Symbol() == OrderSymbol()) && (iMagicNumber != OrderMagicNumber()) )
        {
            // Save order information here
        }
    }
}
```

This compound *if* statement requires that both expressions evaluate to *true* in order to execute the code after the *if* statement. That is, the code inside of the next set of curly braces {} following the *if* statement.

```
if(((OrderSymbol() == Symbol())) && ((OrderMagicNumber() == iMagicNumber )))
{
}
```

The non-exhaustive truth table below shows the logical output for a number of different input variations. Truth tables can be very helpful when defining complex, especially compound, logic.

OrderSymbol	Symbol	OrderMagicNumber	iMagicNumber	<i>if</i> Result	Order Result
GBPUSD	GBPUSD	0	0	True	Included
GBPUSD	EURUSD	1000	0	False	Excluded

iExpert Note

Studies have shown that most people have an easier time evaluating **positive** statements than **negative** statements. Given the choice, try to state your logical requirements in positive language.

EURUSD	EURUSD	0	1000	False	Excluded
EURUSD	GBPUSD	1000	1000	False	Excluded

For example:

(OrderSymbol() == Symbol())

is better than:

(OrderSymbol() != Symbol())

which is better than:

```
!((OrderSymbol() != Symbol() )
```

Now that we have created a *for* loop that iterates through the order pool and have selected the orders the Expert Advisor is interested in, we are ready to copy the values of the order information that will be used later by the Expert Advisor. (Don't be too concerned if you are not sure exactly what values will be needed: adding values to this code is easy and can be done at a later time when the requirement becomes apparent.)

The information that is available for each order is defined in the section *Trade Information Variables*. The values offered by the *Trade Information Variables* are stored in the variables identified in the section *Open Trade Information Variable*.

Now that the *for* loop is set up, the value of the trade information variables can be set.

Trade Information Variables

iTotalTrades

This variable is set just once, at the start of the function. It is set to the result of the MQL function *OrdersTotal*.

```
iTotalTrades = OrdersTotal();
```

dTotalTradeLots

This variable is *added to* each time the loop executes. The value of *dTotalTradeLots* is the summation of all of the lot sizes of all the orders the EA is managing. This MQL code will add the value of *OrderLots* to the variable *dTotalTradeLots*:

```
dTotalTradeLots += OrderLots();
```

This code is equivalent to:

```
dTotalTradeLots = dTotalTradeLots + OrderLots();
```

This code takes the current value stored in dTotalTradeLots, adds it to OrderLots(), and then places the sum back into dTotalTradeLots.

Note: This variable must be explicitly set to zero before the start of the *for* loop. (If not, the starting value will be the last ending value. The number will get large, fast!).

dTotalTradeSwap

This variable is *added to* each time the loop executes. The value of *dTotalTradeSwap* is the summation of the swap amount of all the orders the EA is managing. (The *swap* is the amount of interest added or deducted to an open order each trading day.) This MQL code will add the value of *OrderSwap* to the variable *dTotalTradeSwap*:

```
dTotalTradeSwap += OrderSwap ();
```

This code is equivalent to:

```
dTotalTradeSwap = dTotalTradeSwap + OrderSwap ();
```

Note: This variable must be explicitly set to zero before the start of the *for* loop.

iTotalBuyTrades

This variable is *incremented* each time the loop executes. The value of *iTotalBuyTrades* is the total number of buy positions that the EA is managing. This MQL code will increment the variable *iTotalBuyTrades*:

```
iTotalBuyTrades++;
```

This code is equivalent to:

```
iTotalBuyTrades= iTotalBuyTrades + 1;
```

However, before executing this code, the selected order is verified that is a *buy* order. This is done using the MQL function **OrderType**, the MQL constant **OP_BUY** and an **if** statement:

```
if( OrderType() == OP_BUY )  
    iTotalBuyTrades++;
```

Note: This variable must be explicitly set to zero before the start of the **for** loop.

iTotalSellTrades

This variable is *incremented* each time the loop executes. The value of **iTotalSellTrades** is the total number of sell positions that the EA is managing. This MQL code will increment the variable **iTotalSellTrades**:

```
iTotalSellTrades++;
```

This code is equivalent to:

```
iTotalSellTrades= iTotalSellTrades + 1;
```

Before executing this code, the selected order is verified that is a *sell* order. This is done using the MQL function **OrderType**, the MQL constant **OP_SELL** and an **if** statement:

```
if( OrderType() == OP_SELL )  
    iTotalSellTrades++;
```

Note: This variable must be explicitly set to zero before the start of the **for** loop.

dTotalOpenProfit

This variable is *added to* each time the loop executes. The value of **dTotalOpenProfit** is the summation of the profit amount of all the orders the EA is managing. This MQL code will add the value of **OrderProfit** to the variable **dTotalOpenProfit**:

```
dTotalOpenProfit+= OrderProfit ();
```

This code is equivalent to:

```
dTotalOpenProfit= dTotalOpenProfit + OrderProfit ();
```

Note: This variable must be explicitly set to zero before the start of the **for** loop.

iTotalSelectedTrades

This variable is *incremented* each time the loop executes. The value of **iTotalSelectedTrades** is the total number of all positions that the EA is managing. This MQL code will increment the variable **iTotalSelectedTrades**:

```
iTotalSelectedTrades ++;
```

This code is equivalent to:

```
iTotalSelectedTrades = iTotalSelectedTrades + 1;
```

Note: This variable must be explicitly set to zero before the start of the **for** loop.

The following is the MQL code for the complete *fnGetOpenTradeInfo* function. Notice the variables are set to zero *before* entering the *for* loop.

```
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    dTotalOpenProfit=0;
    dTotalTradeLots=0;
    dTotalTradeSwap=0;
    iTotalBuyTrades=0;
    iTotalSellTrades=0;
    iTotalSelectedTrades=0;
    iTotalTrades=OrdersTotal();

    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos,SELECT_BY_POS)==false)
            continue;
        if( (( OrderSymbol() == Symbol() ) ) && (( OrderMagicNumber() == iMagicNumber ) ) )
        {
            iTotalSelectedTrades++;
            dTotalOpenProfit += OrderProfit();
            dTotalTradeSwap += OrderSwap();
            dTotalTradeLots += OrderLots();
            if( OrderType() == OP_BUY )
                iTotalBuyTrades++;
            if( OrderType() == OP_SELL )
                iTotalSellTrades++;
        }
    }
}
```

fnOpenBuyLogic Function: Using the MQL Function iMA

Module 7: The *fnOpenBuyLogic* Function

Objectives: *Learn to use technical indicators (Moving Average and Relative Strength Indicator).*

Subjects:

- **Ask Price**
- **iMA (Moving Average)**
- **iRSI (Relative Strength Indicator)**
- **Price Constants**
- **The Open Buy Logic Trade Signal**

The *fnOpenBuyLogic* function is responsible for applying the “open buy logic” of the **iStarter** Expert Advisor and setting the *iTradeSignal* variable to *iOpenBuySignal* if the logic evaluates to **true**.

The “open buy logic” can be the application of almost any technical analysis data. For the **iStarter** Expert Advisor, the following “open trade logic” is applied:

Open a buy position when the Ask price is higher than the 12-period exponential moving average and the 14-period RSI indicator is less than 40.

Ask Price

The ask price of the chart’s currency is available from the built-in MQL variable **Ask**.

Moving Average

The moving average is found using the MQL function **iMA**. The signature of the **iMA** function is:

```
double iMA( string symbol, int timeframe, int period, int ma_shift, int ma_method, int applied_price, int shift)
```

The **IMA** function returns a double and requires the following parameters.

Name	Data Type	Description
symbol	string	The symbol of the currency used to calculate the moving average.
timeframe	integer	The time frame of the currency used to calculate the moving average.
period	integer	The averaging period for the calculation. Low numbers are considered a “fast” moving average; high numbers are considered a “slow” moving average.
ma_shift	integer	Shift of value, relative to the indicator’s time frame.
ma_method	integer	The type of moving average requested. See table below.
applied_price	integer	The applied price used in the calculation. See the table below.
shift	integer	Shift of value, in bars, relative to the chart.

Constant variables used to define the moving average method.

Name	Data Type	Description
MODE_SMA	integer	Simple moving average,
MODE_EMA	integer	Exponential moving average,
MODE_SMMA	integer	Smoothed moving average,
MODE_LWMA	integer	Linear weighted moving average

Constant variables used to define the applied price.

Name	Data Type	Description
PRICE_CLOSE	integer	Close price.
PRICE_OPEN	integer	Open price.
PRICE_HIGH	integer	High price.
PRICE_LOW	integer	Low price.
PRICE_MEDIAN	integer	Median price, (high+low)/2.
PRICE_TYPICAL	integer	Typical price, (high+low+close)/3.
PRICE_WEIGHTED	integer	Weighted close price, (high+low+close+close)/4.

As a shortcut, the *symbol* and *timeframe* variables can be set to **NULL** and **zero**, respectively. This will force the indicator to use the symbol and timeframe of the chart to which the Expert Advisor is attached.

The following table summarizes the **iMA** parameter values for the **iStarter** Expert Advisor.

Name	Value	Description
symbol	NULL	Force the indicator to use the symbol of the chart.
timeframe	NULL	Force the indicator to use the timeframe of the chart.
period	12	Uses the last 12 bars to calculate the average.
ma_shift	0	Do not shift the value.
ma_method	MODE_EMA	Calculate using the exponential method.
applied_price	PRICE_CLOSE	Use the close value of each bar for the calculation.
shift	0	Do not shift the value.

To make the MQL code easier to read, declare a *double* variable **dMovingAvg** to hold the value of the **iMA** calculation. The **dMovingAvg** variable can then be used for comparison in an **if** statement.

Note: An indicator does not need to be attached to the chart in order for the Expert Advisor to collect the indicators value. The MQL function will calculate the indicator value without displaying any information on the chart which the Expert Advisor is attached.

iExpert Note

The bar or candle that is currently forming (furthest to the right on a candlestick chart) is numbered zero. Each bar to the left increases in number. A series, or array, of Indicator values also follow this concept of number 0 starting at the far right with each number increasing to the left. The Shift of an indicator value is the index in the array of the value, where shift=0 is the latest value and shift=1 is the value from the last period.

The following MQL code uses the MQL function **iMA** to store the moving average calculation into the **dMovingAvg** variable.

```
double dMovingAvg=0;
dMovingAvg = iMA(NULL, NULL, 12, 0, MODE_EMA, PRICE_CLOSE, 0);
```

Relative Strength Index

The relative strength index is found using the MQL function **IRSI**. The signature of the **IRSI** function is:

```
double iRSI(string symbol, int timeframe, int period, int applied_price, int shift)
```

The **IRSI** function returns a double and requires the following parameters.

Name	Data Type	Description
symbol	string	The symbol of the currency used to calculate the RSI.
timeframe	integer	The time frame of the currency used to calculate the RSI.
period	integer	The number of periods used to calculate the RSI.
applied_price	integer	The applied price used in the calculation.
shift	integer	Shift of value, in bars, relative to the chart.

The following table summarizes the **iRSI** parameter values for the **iStarter** Expert Advisor.

Name	Value	Description
symbol	NULL	Force the indicator to use the symbol of the chart.
timeframe	NULL	Force the indicator to use the timeframe of the chart.
period	12	Uses the last 14 bars to calculate the RSI.
applied_price	PRICE_CLOSE	Use the close value of each bar for the calculation.
shift	0	Do not shift the value.

To make the MQL code easier to read, declare a *double* variable **dRelativeStrength** to hold the value of the **iRSI** calculation. The **dRelativeStrength** variable can then be used for comparison in an **if** statement.

The following MQL code uses the MQL function **iRSI** to store the rsi calculation into the **dRelativeStrength** variable.

iExpert Note

Setting the *applied_price* parameter to any value but **PRICE_OPEN**, and the *shift* parameter to zero will force the indicator calculation to be updated on each incoming tick. This is because the close value of the current bar (shift=0) is not finalized until the bar is actual closed. This provides the latest value but can lead to unexpected results when testing the value of the indicator against a threshold value: the indicator value can quickly move above or below the threshold on each tick. (See Appendix 14 for advanced information.)

```
double dRelativeStrength =0;
dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);
```

The Open Buy Logic

The “open buy logic” of the **iStarter** Expert Advisor Expert Advisor is:

Open a buy position when the Ask price is higher than the 12-period exponential moving average and the 14-period RSI indicator is less than 40.

This is the logic used to set the *iTradeSignal* variable to *iOpenBuySignal*. The following MQL code implements the “open trade logic”.

```
if( (Ask > dMovingAvg) && (dRelativeStrength < 40.0) )
{
    iTradeSignal = iOpenBuySignal;
}
```

The *if* statement evaluates two expressions:

- (Ask > dMovingAvg)
- (dRelativeStrength < 40.0)

Both expression must evaluate *true* (because of the *and (&&)* operator in order for the code in the curly braces after the *if* statement to execute.

The following is the MQL code for the complete *fnOpenBuyLogic* function.

```
void fnOpenBuyLogic()
{
    double dMovingAvg=0;
    dMovingAvg = iMA(NULL,NULL,12,0,MODE_EMA,PRICE_CLOSE,0);

    double dRelativeStrength =0;
    dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);

    if( (Ask > dMovingAvg) && (dRelativeStrength < 40.0) )
    {
        iTradeSignal = iOpenBuySignal;
    }
}
```

fnOpenSellLogic Function: Using the MQL Function *iLowest*

Module 8: The *fnOpenSellLogic* Function

Objective: *Learn to use the series array Low and the MQL function iLowest.*

Subjects Covered:

- Low
- iLowest
- The Open Buy Logic Trade Signal

The *fnOpenSellLogic* function is responsible for applying the “open sell logic” and setting the *iTradeSignal* variable to *iOpenSellSignal* if the logic evaluates to *true*.

The “open sell logic” can be the application of almost any technical analysis data. For the *iStarter* Expert Advisor, the following “open sell logic” is applied:

Open a sell position when the Bid price is lower than the lowest price within the last 18 bars.

Bid Price

The *Bid* price of the chart’s currency is available from the built-in MQL variable *Bid*.

Low Array

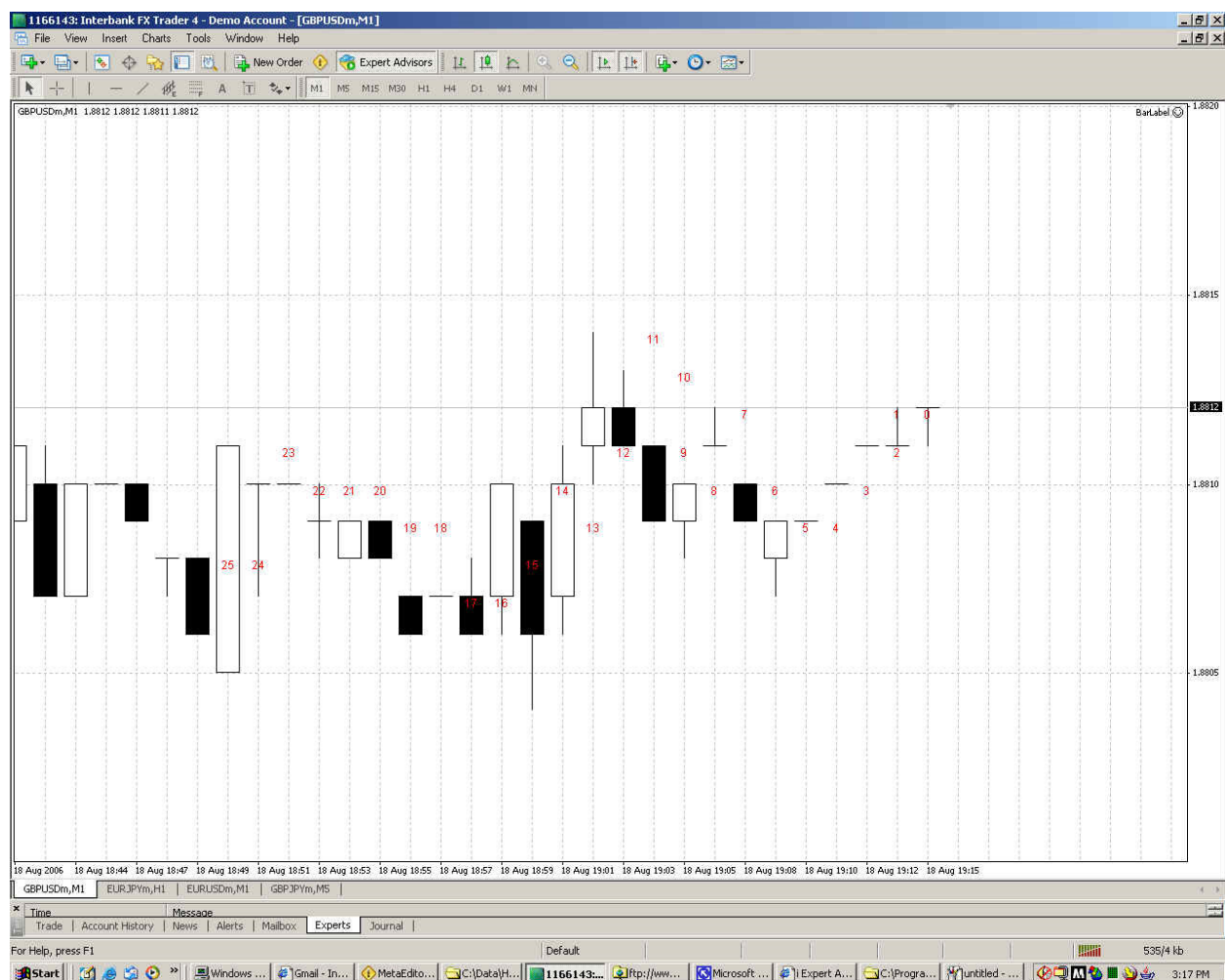
The MQL built-in variable *Low* is an array of the low prices from the price chart to which the Expert Advisor is attached. (Note: The MQL function *iLow* is used to capture the low value from any currency chart. See appendix N)

The *Low* array is a list of variables whose data type is *double*. The list starts at zero, which is the currently forming bar on the chart, and goes back as far as there is data for the chart.

The following table is an example of the contents of the **Low** array.

Index	Low Value (double data type)
0	1.3450
1	1.3456
2	1.3459
3	1.3440
4	1.3454
5	1.3454
6	1.3459
...	...

Note: When viewing a chart, the bar or candle that is currently forming at the far right side of the chart is defined as the bar number zero. Each bar to the left increases by one. All MetaTrader series, such as High, Low, Open and Close follow this same numbering rule. The following price chart shows the candles labeled with their number.



MQL Function *iLowest*

The MQL function *iLowest* is used to find the index of the lowest value within the *Low* array. (Note: *iLowest* returns an *index*, not a *value*.)

The signature of the *iLowest* function is:

```
int iLowest( string symbol, int timeframe, int type, int count=WHOLE_ARRAY,
int start=0)
```

The *iLowest* function returns an integer and requires the following parameters.

Name	Data Type	Description
symbol	string	The symbol of the currency.
timeframe	integer	The time frame of the currency.
type	integer	The series array type. See the table below.
count	integer	The number of values (in direction from the start bar to the back one) on which the calculation is carried out.
start	integer	The index into the array from where to start the calculation.

Constant variables used to define the series array types.

Name	Data Type	Description
MODE_OPEN	integer	Open price.
MODE_LOW	integer	Low price.
MODE_HIGH	integer	High price.
MODE_CLOSE	integer	Close price.
MODE_VOLUME	integer	Volume, used in iLowest() and iHighest() functions.
MODE_TIME	integer	Bar open time, used in ArrayCopySeries() function.

The following table summarizes the *iLowest* parameter values for the *iStarter* Expert Advisor.

Name	Value	Description
symbol	NULL	Force the indicator to use the symbol of the chart.
timeframe	NULL	Force the indicator to use the timeframe of the chart.
type	MODE_LOW	Uses the Low array.
count	18	Look at the last 18 bars.
start	0	Start looking at the first bar in the array (index is zero)

To make the MQL code easier to read, declare an *integer* variable *iLowIndex* to hold the value of the *iLowest* calculation.

```
int iLowestIndex = iLowest(NULL,NULL,MODE_LOW,18,0);
```

Declare a *double* variable *dLowestValue* to hold the value of from the *Low* array. The *dLowestValue* variable can then be used for comparison in an *if* statement.

```
double dLowestValue = Low[iLowestIndex];
```

The following MQL code uses the MQL function *iLowest* to find the index of the lowest value; then the value of the Low array *at that index* is stored into the *dLowestValue* variable.

```
int iLowestIndex = iLowest(NULL,NULL,MODE_LOW,18,0);  
double dLowestValue = Low[iLowestIndex];
```

Note: The equivalent MLQ code can be written in an abbreviated form by placing the *iLowest* function directly into the *Low* array:

```
double dLowestValue = Low[iLowest(NULL,NULL,MODE_LOW,18,0)];
```

The Open Sell Logic

The “open sell logic” of the *iStarter* Expert Advisor Expert Advisor is:

Open a sell position when the Bid price is lower than the lowest price in last 18 bars.

This is the logic used to set the *iTradeSignal* variable to *iOpenSellSignal*. The following MQL code implements the “open sell logic”.

```
if( Bid < dLowestValue )
{
    iTradeSignal = iOpenSellSignal;
}
```

When the expression of the *if* statement evaluates to *true*, the code in the curly braces after the *if* statement will execute, that is, the variable *iTradeSignal* will be set to *iOpenSellSignal*.

The following is the MQL code for the complete *fnOpenSellLogic* function.

```
void fnOpenSellLogic()
{
    int iLowestIndex = iLowest(NULL, NULL, MODE_LOW, 18, 0);
    double dLowestValue = Low[iLowestIndex];

    if( Bid < dLowestValue )
    {
        iTradeSignal = iOpenSellSignal;
    }
}
```

fnCloseBuyLogic Function

Module 9: The *fnCloseBuyLogic* Function

Objective: Use the *iRSI* function to set the *iTradeSignal* variable.

Subjects Covered:

- *iRSI*
- The Close Buy Logic Trade Signal

The *fnCloseBuyLogic* function is responsible for applying the “close buy logic” of the *iStarter* Expert Advisor and setting the *iTradeSignal* variable to *iCloseBuySignal* if the logic evaluates to *true*.

The “close buy logic” can be the application of almost any technical analysis data. For the *iStarter* Expert Advisor, the following “close trade logic” is applied:

Close a buy position when the 14-period RSI indicator is greater than 70.

The Relative Strength Index value is found using the same method as shown in the *fnOpenBuyLogic* function.

The following is the MQL code for the complete *fnOpenSellLogic* function.

```
void fnCloseBuyLogic()
{
    double dRelativeStrength =0;
    dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);

    if( (dRelativeStrength < 40.0) )
    {
        iTradeSignal = iCloseBuySignal;
    }
}
```

fnCloseSellLogic Function

Module 10: The *fnCloseSellLogic* Function

Objective: Use the *iLowest* function to set the *iTradeSignal* variable.

Subjects Covered:

- *iLowest*
- The Close Sell Logic Trade Signal

The *fnCloseSellLogic* function is responsible for applying the “close sell logic” and setting the *iTradeSignal* variable to *iCloseSellSignal* if the logic evaluates to *true*.

The “close sell logic” can be the application of almost any technical analysis data. For the *iStarter* Expert Advisor, the following “close sell logic” is applied:

Close a sell position when the Bid price is higher than the highest price in last 18 bars.

The high value is found using the same method as shown in the *fnOpenSellLogic* function to find the low value.

The following is the MQL code of the complete *fnCloseSellLogic* function.

```
void fnCloseSellLogic()
{
    int iHighestIndex = iHighest(NULL,NULL,MODE_HIGH,18,0);
    double dHighestValue = Low[iHighestIndex];

    if( Bid > dHighestValue )
    {
        iTradeSignal = iCloseSellSignal;
    }
}
```

fnOrder Function: Using the MQL Functions OrderSend and OrderClose

Module 11: The *fnOrder* Function

Objective: *Learn to open an order using OrderSend and close an order using OrderClose.*

Subjects Covered:

- OrderSend
- OrderClose
- The Point Function
- Error Handling

The *fnOrder* function is responsible for checking the value of *iTradeSignal* variable and taking the proper action of opening or closing a position.

The MQL functions used to open and close trading positions are *OrderSend* and *OrderClose*, respectively.

The MQL Function *OrderSend*

The MQL function *OrderSend* is used to open a Buy or Sell position, as well as a pending order. (See appendix 10 for more information about pending orders).

The signature of the *OrderSend* function is:

```
int OrderSend( string symbol, int cmd, double volume, double price, int
slippage, double stoploss, double takeprofit, string comment=NULL, int
magic=0, datetime expiration=0, color arrow_color=CLR_NONE)
```

The **OrderSend** function returns an *integer* and requires the following parameters.

Name	Data Type	Description
symbol	string	The symbol of the currency.
cmd	integer	The type of order to open. See the table below.
volume	double	The number of lots to open.
price	double	The preferred open price.
slippage	integer	The number of points the preferred open price may slip.
stoploss	integer	The stoploss set in the order on the broker's server. This value must be calculated using the price and the currency point value.
takeprofit	integer	The takeprofit set in the order on the broker's server. This value must be calculated using the price and the currency point value.
comment	string	A text comment that can be seen from within the MetaTrader platform when viewing open orders.
magic	integer	A unique number that can be used to identify the trade.
expiration	integer	An expiration time – only used for pending orders.
arrow_color	color	The color of the arrow that will appear on the price chart when the OrderSend function is successful.

The constant variables used to define the command (cmd) types.

Name	Data Type	Description
OP_BUY	integer	Buy (long) position.
OP_SELL	integer	Sell (short) position.
OP_BUYLIMIT	integer	Buy limit pending position.
OP_SELLLIMIT	integer	Sell limit pending position.
OP_BUYSTOP	integer	Buy stop pending position.
OP_SELLSTOP	integer	Sell stop pending position.

Opening a Buy Order

The following table summarizes the **OrderSend** parameter values for opening a Buy position.

Name	Value	Description
symbol	Symbol()	Use the MQL Symbol() function to set the currency symbol.
cmd	OP_BUY	Open a buy position.
volume	dLots	Use the dLots user-defined variable.
price	Ask	Ask is the preferred price used for OP_BUY orders.
slippage	iSlippage	Use the iSlippage user-defined variable.
stoploss	dBuyStopLoss	Use variable dBuyStopLoss, see below.
takeprofit	dBuyTakeProfit	Use variable dBuyStopLoss, see below.
comment	"iStarter"	This can be any value. The name of the Expert Advisor is a good choice.
magic	iMagicNumber	Use the iMagicNumber user-defined variable.
expiration	0	Not used for the OP_BUY command.
arrow_color	Black	Open buy arrows will be black on the chart.

Calculating the Buy Order StopLoss

Calculating the stoploss is a bit complicated for two reasons.

- The value must be multiplied by the **Point** value of the currency.
- The value is an *offset* from the preferred open price.

The stoploss parameter of the **OrderSend** function is the actual price value of the stoploss, such as 1.3409. (The data type is **double**.)

The user-defined variable **iStopLoss** is an absolute value, such as 50. (The data type is **integer**.)

The first step in calculating the stoploss is to convert the **iStopLoss** value into a number of the correct magnitude. This is done by multiplying the value by the built-in MQL predefined variable **Point**. (For the JPY currency pairs the point value is 0.01, for most others the value is 0.0001). The MQL variable **Point** is set to the correct value according to the chart.

Multiplying the variable **iStopLoss** by **Point** simply changes the magnitude of the result:

$$\text{iStopLoss} * \text{Point} = 50 * 0.0001 = 0.0050$$

Now this value can be applied to the preferred open price. For a Buy order, the stoploss is *subtracted* from the preferred open price. (For a Sell order, the stoploss value is *added* to the preferred open price.) The calculation of the **dBuyStopLoss** is:

$$\text{dBuyStopLoss} = \text{Ask} - (\text{iStopLoss} * \text{Point})$$

Calculating the Buy Order TakeProfit

Calculating the takeprofit is a similar to calculating the stoploss.

The takeprofit parameter of the **OrderSend** function is the actual price value of the takeprofit, such as 1.3409. (The data type is **double**.)

The user-defined variable **iTakeProfit** is an absolute value, such as 50. (The data type is **integer**.)

Multiplying the variable **iTakeProfit** by **Point** simply changes the magnitude of the result:

$$\text{iTakeProfit} * \text{Point} = 50 * 0.0001 = 0.0050$$

Now this value can be applied to the preferred open price. For a Buy order, the **iTakeProfit** is *added* to the preferred open price. (For a Sell order, the **iTakeProfit** value is *subtracted* from the preferred open price.) The calculation of the **dBuyTakeProfit** is:

$$\text{dBuyTakeProfit} = \text{Ask} + (\text{iTakeProfit} * \text{Point})$$

The following MQL code is used to open a buy order.

```
double dBuyStopLoss = Ask-(iStopLoss*Point);
double dBuyTakeProfit = Ask+(iTakeProfit*Point);

OrderSend(Symbol(), OP_BUY, dLots, Ask, iSlippage, dBuyStopLoss, dBuyTakeProfit,
"iExpertAdvisor", iMagicNumber, 0, Black);
```

Opening a Sell Order

The following table summarizes the *OrderSend* parameter values for opening a Sell position.

Name	Value	Description
symbol	Symbol()	Use the MQL Symbol() function to set the currency symbol.
cmd	OP_SELL	Open a sell position.
volume	dLots	Use the dLots user-defined variable.
price	Bid	Bid is the preferred price used for OP_SELL orders.
slippage	iSlippage	Use the iSlippage user-defined variable.
stoploss	dSellStopLoss	Use variable dSellStopLoss, see below.
takeprofit	dSellTakeProfit	Use variable dSellTakeProfit, see below.
comment	"iStarter"	This can be any value. The name of the Expert Advisor is a good choice.
magic	iMagicNumber	Use the iMagicNumber user-defined variable.
expiration	0	Not used for the OP_BUY command.
arrow_color	Blue	Open sell arrows will be blue on the chart.

Calculating the Sell Order StopLoss and TakeProfit

The limit values for the Sell order are calculated similar to the Buy order values, except:

- The preferred open price is the **Bid** price.
- The stoploss is added to the **Bid** price.
- The takeprofit is subtracted from the **Bid** price.

The calculation of the dSellStopLoss is:

$$dSellStopLoss = Bid + (iStopLoss * Point)$$

The calculation of the dBuyTakeProfit is:

$$dBuyTakeProfit = Bid - (iTakeProfit * Point)$$

The following MQL code is used to open a sell order.

```
double dSellStopLoss = Bid+(iStopLoss*Point);
double dSellTakeProfit = Bid-(iTakeProfit*Point);

OrderSend(Symbol(), OP_SELL, dLots, Bid, iSlippage, dSellStopLoss, dSellTakeProfit,
"iExpertAdvisor", iMagicNumber, 0, Blue);
```

The MQL Function *OrderClose*

The MQL function *OrderClose* is used to close a Buy or Sell position. (Pending orders are not closed, they are deleted using the MQL function *OrderDelete* . See appendix 10 for more information.)

The signature of the *OrderClose* function is:

```
bool OrderClose( int ticket, double lots, double price, int slippage, color
Color=CLR_NONE)
```

The *OrderClose* function returns a *boolean* (*true* or *false*) and requires the following parameters.

Name	Data Type	Description
ticket	int	The ticket number of the order to close.
lots	double	The number of lots to close.
price	double	The preferred closing price.
slippage	integer	The number of points the preferred open price may slip.
arrow_color	color	The color of the arrow that will appear on the price chart when the OrderClose function is successful.

Closing a Buy Order

The following table summarizes the *OrderClose* parameter values for closing a Buy position.

Name	Value	Description
ticket	OrderTicket()	From within the OrderSelect <i>for</i> loop.
lots	OrderLots()	From within the OrderSelect <i>for</i> loop.
price	Bid	Bid is the preferred closing price used for buy orders.
slippage	iSlippage	Use the iSlippage user-defined variable.
arrow_color	Red	Close arrows will be red on the chart.

The MQL function *OrderClose* requires the order's ticket number as the first parameter. The easiest way to access the order's ticket number is to use a *for* loop similar to the *for* loop used in the *fnGetOpenOrderInfo* function.

This is the MQL source code of the order close *for* loop.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS) == false)
        continue;
    if( (( OrderSymbol() == Symbol())) && (( OrderMagicNumber() == iMagicNumber )) )
    {
        // Add order close logic here
    }
}
```

The following MQL code is used to close a buy order.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;
    if( (( OrderSymbol() == Symbol() ) ) && (( OrderMagicNumber() == iMagicNumber ) ) )
    {
        OrderClose( OrderTicket(), OrderLots(), Bid, iSlippage, Red );
    }
}
```

Closing a Sell Order

The following table summarizes the *OrderClose* parameter values for closing a Sell position.

Name	Value	Description
ticket	OrderTicket()	From within the OrderSelect <i>for</i> loop.
lots	OrderLots()	From within the OrderSelect <i>for</i> loop.
price	Ask	Ask is the preferred closing price used for sell orders.
slippage	iSlippage	Use the iSlippage user-defined variable.
arrow_color	Red	Close arrows will be red on the chart.

The following MQL code is used to close a sell order.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;
    if( (( OrderSymbol() == Symbol() ) ) && (( OrderMagicNumber() == iMagicNumber ) ) )
    {
        OrderClose( OrderTicket(), OrderLots(), Ask, iSlippage, Red );
    }
}
```

The third parameter of the *OrderClose* function, price, is different depending on the order type. The *OrderSelect for* loop tests for the order type using the following code:

```
If ( OrderType() == OP_BUY ) ...
or
If( OrderType == OP_SELL ) ...
```

The following is the *close order* MQL code.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;
    if( ( ( OrderSymbol() == Symbol() ) ) && ( ( OrderMagicNumber() == iMagicNumber ) ) )
    {
        if( OrderType() == OP_BUY )
            OrderClose( OrderTicket(), OrderLots(), Bid, iSlippage, Red );

        if( OrderType() == OP_SELL )
            OrderClose( OrderTicket(), OrderLots(), Ask, iSlippage, Red );
    }
}
```

Error Handling

The MQL functions *OrderSend* and *OrderClose* return values that may indicate an error has occurred. Specific error handling is outside the scope of this book; however, even a simple Expert Advisor should at least notify the user that an error has occurred.

The *OrderSend* function returns an *integer* data type. If the function succeeds, it returns a positive number: the number of the order's ticket. If the function fails it returns a negative number. The *OrderSend* function can be tested for failure by checking if the return value is less than zero.

The *OrderClose* function returns a *boolean* value (*true* or *false*). If the function succeeds, it returns *true*. If the function fails, it returns *false*. The *OrderClose* function can be tested for failure by checking if the return value is equal to *false*.

The MQL Functions *GetLastError*, *ErrorDescription* and *Print*

The MQL function *GetLastError* is used to capture the last error that has occurred within the Expert Advisor. The signature of *GetLastError* is:

```
int GetLastError()
```

The function *GetLastError* returns an *integer* and requires no parameters. The *integer* error numbers returned by *GetLastError* are not defined in the MQL platform. These numbers must be explicitly included in the Expert Advisor MQL file using the *include* directive. The numbers are defined in the file *stderr.mqh* (This file is part of the MetaTrader platform). This code should be inserted in the MQL file *before* the reference to the *GetLastError* function. Normally, all *include* directives are placed at the beginning of the MQL file. The MQL source code to include the *stderr.mqh* file is:

```
#include <stderr.mqh>
```

The integer returned by the *GetLastError* function is one of the predefined MQL error constants (See appendix 6 for the full table). This number is helpful, but there is another MQL function that will return a descriptive error message. This function is called *ErrorDescription*.

The signature of *ErrorDescription* is:

```
string ErrorDescription(int errno)
```

The function *ErrorDescription* returns a *string* and requires an *integer* parameter. The *string* error messages returned by *ErrorDescription* are not defined in the MQL platform. These strings must be explicitly included in the Expert Advisor MQL file using the *include* directive. The strings are defined in the file `stdlib.mqh` (This file is part of the MetaTrader platform). This code should be inserted in the MQL file *before* the reference to the *ErrorDescription* function. Normally, all include directives are placed at the beginning of the MQL file. The MQL source code to include the `stdlib.mqh` file is:

```
include <stdlib.mqh>
```

The MQL function *Print* is used to print information to the Expert Advisor log file. This log file can be viewed in the *Experts* tab on the MetaTrader terminal platform. (There many ways to send information out of an Expert Advisor see appendix 15 for more information).

The signature of the *Print* function is:

```
void Print(...)
```

The *Print* function returns no value and accepts up to 64 parameters that can be *integer*, *double* or *string* data type.

The following MQL code sample sends an error message to the experts log if the *OrderClose* function fails.

```
bool bOrderCloseStatus;  
string strErrorMessage;  
int iErrorNumber;  
  
bOrderCloseStatus = OrderClose( OrderTicket(), OrderLots(), Bid, iSlippage, Red );  
  
if( bOrderCloseStatus == false )  
{  
    iErrorNumber = GetLastError();  
    strErrorMessage = ErrorDescription(iErrorNumber);  
    Print( "Order Close Failed: ", strErrorMessage );  
}
```

The error handling functionality requires declares these variables.

Variable Name	Data Type	Usage
bOrderCloseStatus	boolean	Stores the return value of the OrderClose function.
strErrorMessage	string	Stores the return value of the ErrorDescription function.
iErrorNumber	integer	Stores the return value of the GetLastError function.

If the *OrderClose* message fails by returning the value *false*, the error is captured and sent to the *Experts* log using the *Print* function.

The following MQL code sample sends an error message to the *Experts* log if the *OrderSend* function fails.

```
int iOrderOpenStatus;  
string strErrorMessage;  
int iErrorNumber;  
  
iOrderOpenStatus = OrderSend(Symbol(), OP_BUY, dLots, Ask, iSlippage, dBuyStopLoss,  
dBuyTakeProfit, "iExpertAdvisor", iMagicNumber, 0, Black);  
  
if( iOrderOpenStatus < 0 )  
{  
    iErrorNumber = GetLastError();  
    strErrorMessage = ErrorDescription(iErrorNumber);  
    Print( "OrderSend Failed: ", strErrorMessage );  
}
```

This MQL code declares the additional *integer* variable *iOrderOpenStatus* to store the return value of the *OrderSend* function.

The following is the MQL code of the complete *fnOrder* function (line numbers are included).

```
1. void fnOrder()
2. {
3.     int iOrderOpenStatus;
4.     string strErrorMessage;
5.     int iErrorNumber;
6.     bool bOrderCloseStatus;

7.     if( (iTradeSignal == iOpenBuySignal) && (iMaxTrades < iTotalSelectedTrades) )
8.     {
9.         double dBuyStopLoss = Ask-(iStopLoss*Point);
10.        double dBuyTakeProfit = Ask+(iTakeProfit*Point);
11.        iOrderOpenStatus = OrderSend(Symbol(),
            OP_BUY,dLots,Ask,iSlippage,dBuyStopLoss,dBuyTakeProfit,"iExpertAdvisor",iMagicNumber,0,Black);
12.        if( iOrderOpenStatus < 0 )
13.        {
14.            a. iErrorNumber = GetLastError();
15.            b. strErrorMessage = ErrorDescription(iErrorNumber);
16.            c. Print( "OrderSend Failed: ", strErrorMessage );
17.        } // end of if iOrderOpenStatus
18.        return;
19.    } // end of if iOpenBuySignal

20.    if( (iTradeSignal == iOpenSellSignal) && (iMaxTrades < iTotalSelectedTrades) )
21.    {
22.        double dSellStopLoss = Bid+(iStopLoss*Point);
23.        double dSellTakeProfit = Bid-(iTakeProfit*Point);
24.        OrderSend(Symbol(),OP_SELL,dLots,Bid,iSlippage,dSellStopLoss,dSellTakeProfit,"iExpertAdvisor",iMagicNumber
            ,0,Blue);
25.        if( iOrderOpenStatus < 0 )
26.        {
27.            a. iErrorNumber = GetLastError();
28.            b. strErrorMessage = ErrorDescription(iErrorNumber);
29.            c. Print( "OrderSend Failed: ", strErrorMessage );
30.        } // end of if iOrderOpenStatus
31.        return;
32.    } // end of if iOpenSellSignal

33.    if( (iTradeSignal == iCloseBuySignal) || (iTradeSignal == iCloseBuySignal) )
34.    {
35.        iTotalTrades = OrdersTotal();
36.        for(int pos=0;pos<iTotalTrades;pos++)
37.        {
38.            a. if(OrderSelect(pos,SELECT_BY_POS)==false)
39.                i. continue;

40.            b. if( ( OrderSymbol() == Symbol() ) ) && ( OrderMagicNumber() == iMagicNumber ) )
41.            {
42.                c. {
43.                    i. if( OrderType() == OP_BUY )
44.                    {
45.                        ii. bOrderCloseStatus = OrderClose( OrderTicket(), OrderLots(), Bid, iSlippage, Red );
46.                        iii. if( bOrderCloseStatus == false )
47.                        {
48.                            1. iErrorNumber = GetLastError();
49.                            2. strErrorMessage = ErrorDescription(iErrorNumber);
50.                            3. Print( "OrderClose Failed: ", strErrorMessage );
51.                        } // end of if bOrderCloseStatus
52.                        vii. } // end of if OP_BUY

53.                    viii. if( OrderType() == OP_SELL )
54.                    {
55.                        ix. bOrderCloseStatus = OrderClose( OrderTicket(), OrderLots(), Ask, iSlippage, Red );
56.                        x. if( bOrderCloseStatus == false )
57.                        {
58.                            1. iErrorNumber = GetLastError();
59.                            2. strErrorMessage = ErrorDescription(iErrorNumber);
60.                            3. Print( "OrderClose Failed: ", strErrorMessage );
61.                        } // end of if bOrderCloseStatus
62.                        xiv. } // end of if OP_SELL
63.                        xv. return;
64.                    } // end of if symbol
65.                } // end of OrderSelect for loop
66.            } // end of if iCloseBuySignal
67.        } // end of fnOrder
```

The following observations are made of the *fnOrder* function.

- Line 7. This is the logical test for opening a Buy order. The signal must be set and the number of trades already opened by this Expert Advisor must be less than the maximum allowed.
- Lines 9-10. This is the calculation of the limits for the Buy order.
- Line 13a-c. This is the error handling of a failed *OrderSend*.
- Line 15. If there was an attempt to open a Buy order, regardless of the status of the *OrderSend* function, the function should return and the Expert Advisor should exit. The MQL documentation states an Expert Advisor should wait at least 5 seconds after sending an order request to the server before sending any further requests. Exiting from the Expert Advisor after the call to *OrderSend* is normally a sufficient wait time.
- Line 17. This is the logical test for opening a Sell order. The signal must be set and the number of trades already opened by this Expert Advisor must be less than the maximum allowed.
- Lines 9-10. This is the calculation of the limits for the Sell order.
- Line 25. If there was an attempt to open a Sell order, regardless of the status of the *OrderSend* function, the function should return and the Expert Advisor should exit. The MQL documentation states an Expert Advisor should wait at least 5 seconds after sending an order request to the server before sending any further requests. Exiting from the Expert Advisor after the call to *OrderSend* is normally a sufficient wait time.
- Line 27. This is the logical test for closing a Buy or Sell order.
- Line 30. This is the start of the *OrderSelect* for loop.
- Line 31.b.iv -vi. This is the error handling of a failed *OrderClose* for a Buy order.
- Line 31.b.ii -xiv. This is the error handling of a failed *OrderClose* for a Sell order.
- Line 30.c.xv. If there was an attempt to close a Buy order, regardless of the status of the *OrderSend* function, the function should return and the Expert Advisor should exit. The MQL documentation states an Expert Advisor should wait at least 5 seconds after sending an order request to the server before sending any further requests. Exiting from the Expert Advisor after the call to *OrderSend* is normally a sufficient wait time.

Building the *iStarter* Expert Advisor

Module 12: Building the *iStarter* Expert Advisor

Objectives: *Build the iStarter Expert Advisor.*

Subjects:

- **Comments**
- **Properties**
- **Start Function**
- **Syntax Errors**
- **Compiling the MQL Code**

The Expert Advisor developed for this book is named ***iStarter***. Generally speaking the MQL code of a typical Expert Advisor, including ***iStarter***, follows this order:

- Comments about the Expert Advisor that would be useful to a person reading the MQL code.
- Property definitions supported by the MQL compiler.
- Include directives to include additional MQL source files.
- The declarations of extern variables (user-settable variables).
- The declarations of variables need by all functions in the Expert Advisor.
- The contents of the **init** function.
- The contents of the **start** function.
- The contents of the **deinit** function.
- All user defined functions.

Comments in an MQL File

Comments are textual statements within an MQL code that are ignored by the MQL compiler. There are two methods of inserting comments into a file. The first is applied to a single line only using two forward slash (//) characters.

```
// This line is commented out.
```

The second allows more than one line to be commented out. The start sequence is a forward slash and an asterisk; the end sequence is an asterisk and a forward slash.

```
/* Both of these lines are  
commented out.  
*/
```

If the development of the Expert Advisor was started using the Expert Advisor Wizard, the beginning of the MQL file will have the following information.

```
//+-----+
//|                                     iExpertAdvisor.mq4 |
//|                                     Copyright © 2007, MetaQuotes Software Corp. |
//|                                     http://www.metaquotes.net |
//+-----+
```

Property Definitions

MQL defines a set of Property definitions. These are a set of special commands that drive the MQL compiler to execute specific behavior.

Most of the commands are related to building custom indicators, which are outside the scope of this book, but there two available for Expert Advisors that may be useful.

The [link](#) and [copyright](#) properties will cause the assigned text to be shown in different locations on the Expert Advisor (exactly where depends on the version of the MetaTrader Platform).

This is the syntax of the [link](#) and [copyright](#) property commands:

#property	link	"http://www.iExpertAdvisor.com"
#property	copyright	"iExpertAdvisor, LLC 2008"

Including External Code

The MQL #include Directive

MQL files can be added to another MQL file using the special [include](#) directive. The syntax of the [include](#) directive is:

```
#include <stderror.mqh>
```

A file with an "mqh" extension is called a MQL header file. It usually contains variable declarations. In this example, the file "stderror.mqh", contains the following list of definitions.

```
#define ERR_NO_ERROR 0
#define ERR_NO_RESULT 1
#define ERR_COMMON_ERROR 2
#define ERR_INVALID_TRADE_PARAMETERS 3
#define ERR_SERVER_BUSY 4
#define ERR_OLD_VERSION 5
#define ERR_NO_CONNECTION 6
```

The MQL **#define** directive is used to define constant declarations. The trade signal variables in the **iStarter** Expert Advisor could have been declared as constant declarations using the **#define** directive.

By convention, constant declarations names are usually upper case. For example:

```
#define      OPEN_BUY_SIGNAL      100
#define      OPEN_SELL_SIGNAL     200
#define      CLOSE_BUY_SIGNAL     300
#define      CLOSE_SELL_SIGNAL    400
```

The **#define** directive can be used to declare any type, including **strings**:

```
#define      EXPERT_NAME          "iStarter"
#define      VERSION              "1.0"
```

The MQL **#import** Directive

In addition to including MQL source code using the **#include** directive, code from compiled MQL files (*.ex4 files) and operating system files (*.dll files) can be imported using the **#import** directive. The syntax of the import directive is:

```
#import "stdlib.ex4"
string ErrorDescription(int error_code);
```

The function signatures used by the Expert Advisor are listed directly after the **import** command.

Declaration of Variables

The phrase “variable declaration” is used to describe the process of defining a variable data type, name and optionally the default value. Also, the variable declaration may include modifiers such as *extern* and *static*.

A variable declaration follows this syntax:

modifier data type name = value;

For example:

extern int iStopLoss = 25;

The *extern* modifier allows the variable to be set from the *inputs* tab when starting the Expert Advisor.

The *static* modifier is used for variables that are declared within a user-defined function. The *static* modifier causes the value of the variable to be saved when the function exits. This allows the function to retrieve the value of the variable from the last time the Expert Advisor was invoked. Alternatively, the variable can be declared outside of all functions at the start of the Expert Advisor.

The *init* Function

The MQL *init* function is not defined in the *iStarter* Expert Advisor. It could be used to declare the start time of the Expert Advisor, or the starting *Bar*.

```
datetime iStartTime;  
int iStartBar;  
  
int init()  
{  
    iStartTime = TimeCurrent();  
    iStartBar = Bars;  
}
```

The *start* Function

The heart of the *iStarter* Expert Advisor is comprised of the following functions.

- fnGetOpenTradeInfo
- fnOpenBuyLogic
- fnOpenSellLogic
- fnCloseBuyLogic
- fnCloseSellLogic
- fnOrder

These functions will all be called from the entry point of the Expert Advisor, the *start* function.

```

int start()
{
    fnGetOpenTradeInfo();
    fnOpenBuyLogic();
    fnOpenSellLogic();
    fnCloseBuyLogic();
    fnCloseSellLogic();
    fnOrder();

    return(0);
}

```

The *deinit* Function

The MQL ***deinit*** function is not defined in the ***iStarter*** Expert Advisor. It is called whenever the Expert Advisor exits. It could be used to set variables or perform specific cleanup tasks. The reason the Expert Advisor is exiting can be found using the MQL function ***UninitializeReason***. (See appendix 11 for more information.)

The User Defined Functions

The user-defined functions of the ***iStarter*** Expert Advisor are the six functions developed in the previous modules: ***fnGetOpenTradeInfo***, ***fnOpenBuyLogic***, ***fnOpenSellLogic***, ***fnCloseBuyLogic***, ***fnCloseSellLogic***, and ***fnOrder***.

Expert Advisor Files

The MQL source file is created using the MetaEditor as a text editor. The file's name is ***iStarter.mq4***.

The MetaEditor will compile the file ***iStarter.mq4*** file and create the ***iStarter.ex4*** file, if the compilation is successful.

The ***iStarter.ex4*** file is the Expert Advisor. This is the file that is attached to a currency chart in the MetaTrader platform.

MQL Syntax Errors, Debugging and Testing

If you spend enough time writing MQL files, you are sure to create a few syntax errors. A syntax error simply means the MQL code is not correctly formatted. Syntax errors appear in the ***Errors*** tab on the bottom of the MetaEditor.

Fortunately, the MetaEditor usually generates a very specific error message along with the exact line number of the offending line. Also, double-clicking the error message in the ***Error*** tab will bring the cursor directly to the syntax error.

One of the more difficult syntax errors to correct are the mismatched parenthesis () or mismatched braces { } error. The best way to locate these errors is to temporarily comment out sections of code

(using `/* */`) until the error message is no longer generated. Then slowly bring the code back in so that you can locate the exact location of the error.

There is a significant difference between *debugging* a trading system and *testing* a trading system. Debugging is performed when the trading system does not follow the rules the developer intended the system to follow. This is usually due to logical errors or a lack of knowledge regarding an MQL function. Testing of a trading system can only begin *after* it has been fully debugged. Testing a trading system is performed to find the most profitable method of configuring and running the system.

Debugging an expert advisor can usually be done using the MetaTrader platform's Strategy Tester.

The MQL function **Print** is helpful when debugging an Expert Advisor. Simply place **Print** functions, with relevant information, throughout the Expert Advisor, especially where important logic is evaluated. The information from the **Print** functions is sent to the *Experts* tab in the MetaTrader terminal and the *Journal* tab from within the Strategy Tester.

In the **iStarter** Expert Advisor, it is useful to print the values of the **iTradeSignal** variable as well as the variables used when applying the logic to determine the trade signal (indicator variables, **Ask**, **High**, etc). These **Print** statements can easily be removed later, if desired, after the behavior of the Expert Advisor has been verified.

The framework of the **iStarter** Expert Advisor was developed for two primary reasons:

1. The logic is easy to debug. A single variable, **iTradeSignal**, determines the behavior of the system. Debugging the system is simple: simply understand how and when the **iTradeSignal** is set to different values.
2. Changes are easy to make. All changes are made to one of the six functions. Most changes are limited to one or two functions. An MQL programmer can be confident changes made in one function will not affect code logic in the other functions.

The Complete *iStarter* Expert Advisor Code

```
//+-----+
//|                                     iStarter.mq4 |
//|                                     Copyright © 2008, iExpertAdvisor, LLC |
//|                                     http://www.iExpertAdvisor.com |
//+-----+
#property copyright "Copyright © 2008, iExpertAdvisor, LLC"
#property link      "http://www.iExpertAdvisor.com"

/*
Expert Advisor MQL code to accompany book:
MQL for Traders: Learn to Build a MetaTrader Expert Advisor in One Day
David M. Williams
Copyright © 2008 All Rights Reserved
*/

// Include files for error codes and error messages
#include <stderror.mqh>
#include <stdlib.mqh>

// User Set-able Variables
extern double dLots=1.0;
extern double iStopLoss=50;
extern double iTakeProfit=50;
extern int iSlippage=3;
extern int iMaxTrades=1;
extern int iMagicNumber=1000;

// Trade Signal Variables
int iTradeSignal=0;
int iOpenBuySignal=100;
int iCloseBuySignal=-200;
int iOpenSellSignal=300;
int iCloseSellSignal=-400;
int iNoSignal=-1;

// Open Trade Information Variables
int iLastTradeType;
int iTotalTrades;
double dTotalTradeLots;
double dTotalTradeSwap;
int iTotalBuyTrades;
int iTotalSellTrades;
double dLastTradeOpenPrice;
int iLastTradeOpenTime;
double dLastTradeStopLoss;
double dLastTradeTakeProfit;
double dLastTradeType;
double dLastTradeTicket;
string strLastTradeSymbol;
string strLastTradeComment;
double dTotalOpenProfit;
int iTotalSelectedTrades;
```

iStarter MQL code (continued)

```
// The init function, only execute the first time the Expert Advisor is attached to a chart
int init()
{
    iLastTradeType=1;
}

// The entry point of the Expert Advisor: the start function
int start()
{
    fnGetOpenTradeInfo();
    fnOpenBuyLogic();
    fnOpenSellLogic();
    fnCloseBuyLogic();
    fnCloseSellLogic();
    fnOrder();
    fnTrailingStop();
    return(0);
}

// The user-defined functions
// Get information about all open trades
void fnGetOpenTradeInfo()
{
    dTotalOpenProfit=0;
    dTotalTradeLots=0;
    iTotalBuyTrades=0;
    iTotalSellTrades=0;
    dTotalTradeSwap=0;
    iTotalSelectedTrades=0;
    iTotalTrades=OrdersTotal();
    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos,SELECT_BY_POS)==false)
            continue;
        if( ( OrderSymbol() == Symbol() ) ) && ( OrderMagicNumber() == iMagicNumber ) )
        {

            iTotalSelectedTrades++;
            dTotalOpenProfit += OrderProfit();
            dTotalTradeSwap += OrderSwap();
            dTotalTradeLots += OrderLots();
            if( OrderType() == OP_BUY )
            {
                iTotalBuyTrades++;
                iLastTradeType = OP_BUY;
            }
            if( OrderType() == OP_SELL )
            {
                iTotalSellTrades++;
                iLastTradeType=OP_SELL;
            }
        }
    }
}
```


iStarter MQL code (continued)

```
// The logic for opening a Buy order.
// This will set the variable iTradeSignal to iOpenBuySignal if the logic evaluates to true.
void fnOpenBuyLogic()
{
    double dMovingAvg=0;
    dMovingAvg = iMA(NULL,NULL,12,0,MODE_EMA,PRICE_CLOSE,0);

    double dRelativeStrength =0;
    dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);

    if( (Ask > dMovingAvg) && (dRelativeStrength > 40.0) )
    {
        if( (iLastTradeType == OP_SELL) || (iLastTradeType == -1) )
            iTradeSignal = iOpenBuySignal;
    }
}

// The logic for opening a Sell order.
// This will set the variable iTradeSignal to iOpenSellSignal if the logic evaluates to true.
void fnOpenSellLogic()
{
    int iLowestIndex = iLowest(NULL,NULL,MODE_LOW,18,0);
    double dLowestValue = Low[iLowestIndex];

    if( Bid < dLowestValue )
    {
        if( (iLastTradeType == OP_BUY) || (iLastTradeType == -1) )
            iTradeSignal = iOpenSellSignal;
    }
}

// The logic for closing a Buy order.
// This will set the variable iTradeSignal to iCloseBuySignal if the logic evaluates to true.
void fnCloseBuyLogic()
{
    double dRelativeStrength =0;
    dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);

    if( (dRelativeStrength < 40.0) )
    {
        iTradeSignal = iCloseBuySignal;
    }
}

// The logic for closing a Sell order.
// This will set the variable iTradeSignal to iCloseSellSignal if the logic evaluates to true.
void fnCloseSellLogic()
{
    int iHighestIndex = iHighest(NULL,NULL,MODE_HIGH,18,0);
    double dHighestValue = Low[iHighestIndex];

    if( Bid > dHighestValue )
    {
        iTradeSignal = iCloseSellSignal;
    }
}
```

iStarter MQL code (continued)

```
// The code for opening or closing orders.
// This will execute according to the value of the variable iTradeSignal.
void fnOrder()
{
    int iOrderOpenStatus;
    string strErrorMessage;
    int iErrorNumber;
    bool bOrderCloseStatus;

    if( (iTradeSignal == iOpenBuySignal) && (iMaxTrades < iTotalSelectedTrades) )
    {
        double dBuyStopLoss = Ask-(iStopLoss*Point);
        double dBuyTakeProfit = Ask+(iTakeProfit*Point);
        iOrderOpenStatus =
        OrderSend(Symbol(), OP_BUY, dLots, Ask, iSlippage, dBuyStopLoss, dBuyTakeProfit, "iExpertAdvisor", iMagicNumber, 0, Black);
        if( iOrderOpenStatus < 0 )
        {
            iErrorNumber = GetLastError();
            strErrorMessage = ErrorDescription(iErrorNumber);
            Print( "OrderSend Failed: ", strErrorMessage );
        } // end of if iOrderOpenStatus
        return;
    } // end of if iOpenBuySignal

    if( (iTradeSignal == iOpenSellSignal) && (iMaxTrades < iTotalSelectedTrades) )
    {
        double dSellStopLoss = Bid+(iStopLoss*Point);
        double dSellTakeProfit = Bid-(iTakeProfit*Point);
        OrderSend(Symbol(), OP_SELL, dLots, Bid, iSlippage, dSellStopLoss, dSellTakeProfit, "iExpertAdvisor", iMagicNumber, 0, Blue);
        if( iOrderOpenStatus < 0 )
        {
            iErrorNumber = GetLastError();
            strErrorMessage = ErrorDescription(iErrorNumber);
            Print( "OrderSend Failed: ", strErrorMessage );
        } // end of if iOrderOpenStatus
        return;
    } // end of if iOpenSellSignal

    if( (iTradeSignal == iCloseBuySignal) || (iTradeSignal == iCloseBuySignal) )
    {
        iTotalTrades = OrdersTotal();
        for(int pos=0; pos<iTotalTrades; pos++)
        {
            if(OrderSelect(pos, SELECT_BY_POS) == false)
                continue;

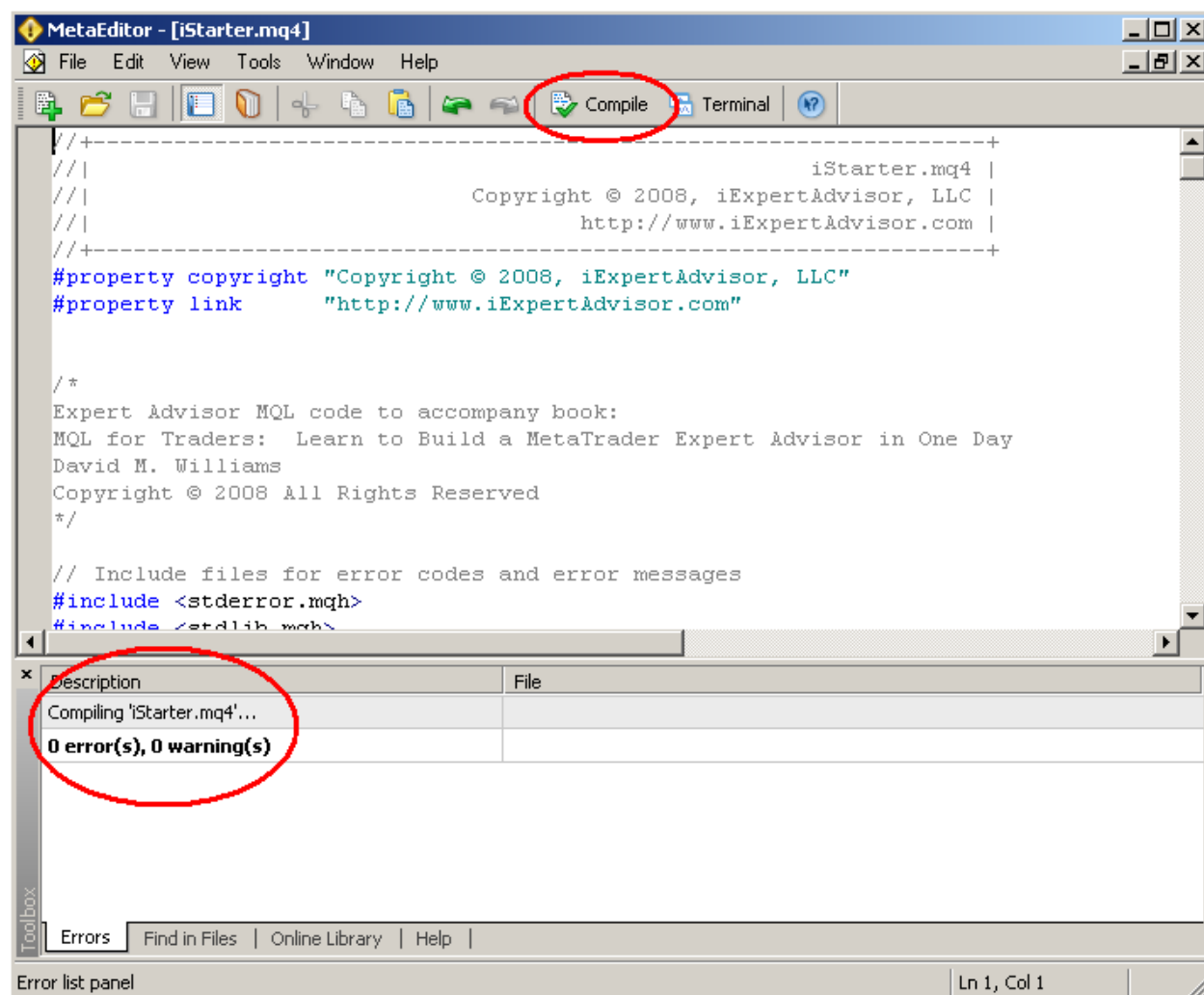
            if( ( OrderSymbol() == Symbol() ) && ( OrderMagicNumber() == iMagicNumber ) )
            {
                if( OrderType() == OP_BUY )
                {
                    bOrderCloseStatus = OrderClose( OrderTicket(), OrderLots(), Bid, iSlippage, Red );
                    if( bOrderCloseStatus == false )
                    {
                        iErrorNumber = GetLastError();
                        strErrorMessage = ErrorDescription(iErrorNumber);
                        Print( "OrderClose Failed: ", strErrorMessage );
                        return;
                    } // end of if bOrderCloseStatus
                } // end of if OP_BUY

                if( OrderType() == OP_SELL )
                {
                    bOrderCloseStatus = OrderClose( OrderTicket(), OrderLots(), Ask, iSlippage, Red );
                    if( bOrderCloseStatus == false )
                    {
                        iErrorNumber = GetLastError();
                        strErrorMessage = ErrorDescription(iErrorNumber);
                        Print( "OrderClose Failed: ", strErrorMessage );
                        return;
                    } // end of if bOrderCloseStatus
                } // end of if OP_SELL
            } // end of if symbol
        } // end of OrderSelect for loop
    } // end of if iCloseBuySignal
} // end of fnOrder
```

Compiling the *iStarter* Expert Advisor

Compiling the *iStarter* Expert Advisor is easy.

- Start the MetaEditor application.
- Open the file “iStarter.mq4”.
- Click the **Compile** button near the top of the MetaEditor window.
- View the **Errors** tab to see the status of the compilation.
- If there are no errors, the Expert Advisor file (iStarter.ex4) has been built.



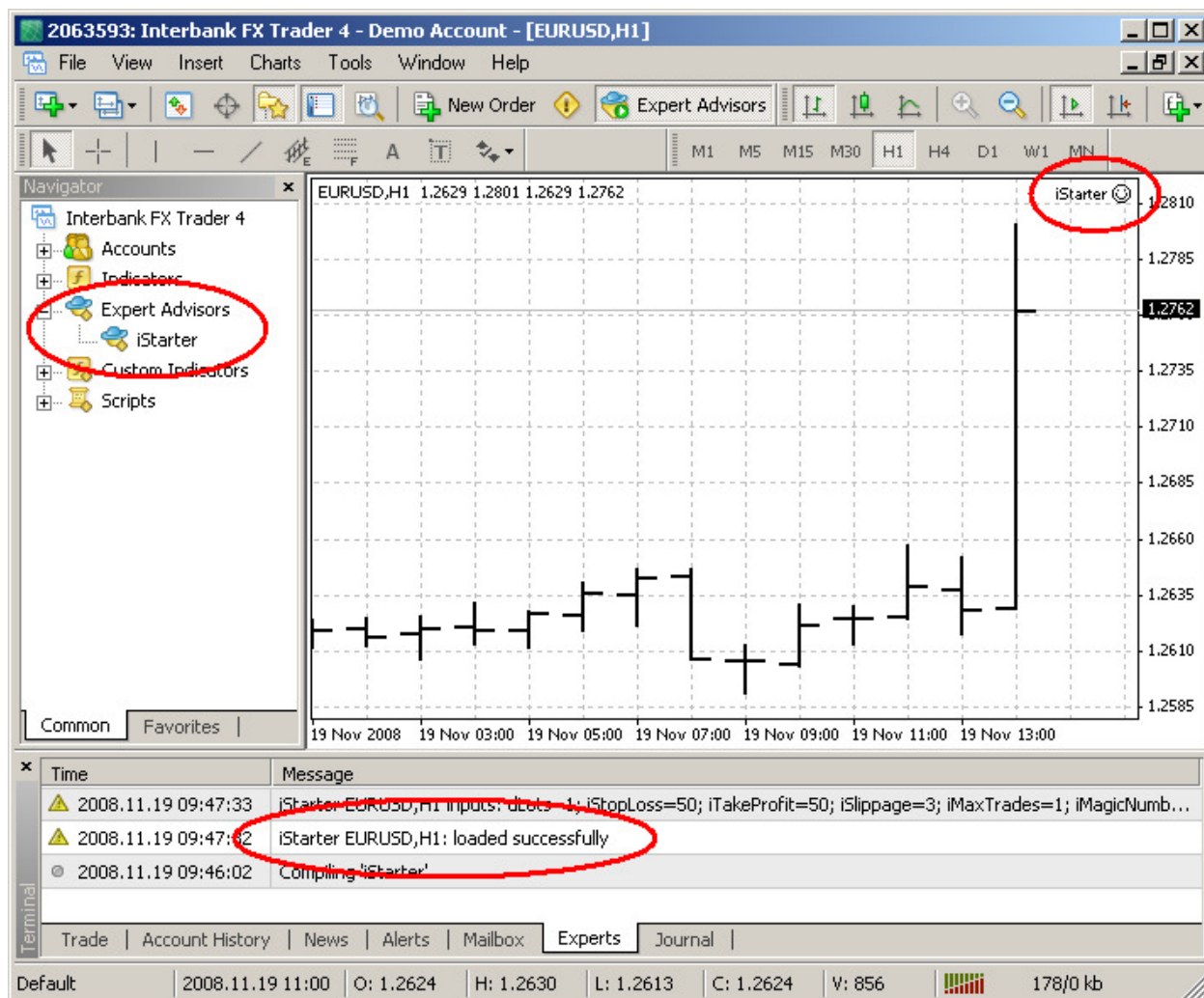
Running the *iStarter* Expert Advisor

After the *iStarter* Expert Advisor has been successfully built, it can be seen from the MetaTrader Terminal's **Navigator** window, under the "Expert Advisors" menu.

The *iStarter* Expert Advisor can be attached to a price chart by double clicking the "iStarter" menu item.

When an Expert Advisor has been attached to a chart and is running, the name of the Expert Advisor will be shown in the top right corner along with a smiley face.

The **Experts** tab, along the bottom of the window, will indicate that the Expert Advisor has been successfully loaded.



Slightly Advanced Topics

Module 13: Advanced Topics

Objective: *Introduce topics to enhance the functionality of the iStarter Expert Advisor.*

Subjects:

- **Trailing Stops: Using OrderModify**
- **Multiple Trades**
- **Controlling Trades Openings**

Trailing Stop

It's hard to consider a trailing stop to be an advanced topic: it is a feature that almost all traders want in their Expert Advisor. The trailing stop code can easily be added to the **iStarter** Expert Advisor by creating a new user-defined function and adding to the Expert Advisor's **start** function.

A trailing stop reads the stoploss of an open trade on every incoming tick, and if the price of the currency has moved away from the stop by a predefined number of points, the stoploss is modified.

The MQL function **OrderModify** is used to modify the parameters of a Buy or Sell position, as well as a pending order. (See appendix 10 for more information about pending orders).

The signature of the **OrderModify** function is:

```
bool OrderModify( int ticket, double price, double stoploss, double  
takeprofit, datetime expiration, color arrow_color=CLR_NONE)
```

The **OrderModify** function returns a boolean value and requires the following parameters.

Name	Data Type	Description
ticket	int	The ticket number of the order.
price	double	The preferred open price.
stoploss	integer	The stoploss set in the order on the broker's server. This value must be calculated using the price and the currency point value.
takeprofit	integer	The takeprofit set in the order on the broker's server. This value must be calculated using the price and the currency point value.
expiration	integer	An expiration time – only used for pending orders.
arrow_color	color	The color of the arrow that will appear on the price chart when the OrderSend function is successful.

The MQL function **OrderModify** requires the order's ticket number as the first parameter. The easiest way to access the order's ticket number is to use a *for* loop similar to the *for* loop used in the fnGetOpenOrderInfo function and the fnOrder function.

Modifying a Buy Order

The following table summarizes the **OrderModify** parameter values for modifying a Buy position.

Name	Value	Description
ticket	OrderTicket()	From within the OrderSelect <i>for</i> loop.
price	OrderOpenPrice()	From within the OrderSelect <i>for</i> loop.
stoploss	dTrailingStop	Use the dTrailingStop. See below.
takeprofit	OrderTakeProfit()	From within the OrderSelect <i>for</i> loop.
expiration	0	An expiration time – only used for pending orders.
arrow_color	Green	The color of the arrow that will appear on the price chart when the OrderSend function is successful.

Calculating the Buy Order Trailing Stop

The actual value of the trailing stop is usually declared as an extern variable so the user can set the value when starting the Expert Advisor.

```
extern int iTrailingStop=10;
```

This *iTrailingStop* variable is an integer data type. It is the number of points of a *profitable* trade's stoploss – adjusted at each incoming tick.

The value of the *dTrailingStop* variable is the newly calculated stoploss. It is calculated similar to the calculation of the stoploss for the OrderSend function.

```
dTrailingStop = Ask – (iTrailingStop *Point)
```

The Buy order is checked if it is profitable and if the current stoploss is further away than *iTrailingStop* before applying this new stoploss.

The order is checked for profitability using the MQL function *OrderProfit*.

```
If( OrderProfit() > 0 )
```

The order current stoploss is checked using the following code.

```
If( OrderStopLoss() < (Ask – iTrailingStop*Point) )
```

The following MQL code is used to modify the stoploss of a buy order.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;

    if( ( ( OrderSymbol() == Symbol() ) ) && ( ( OrderMagicNumber() == iMagicNumber ) ) )
    {
        if( OrderType() == OP_BUY )
        {
            dTrailingStop = Ask - (iTrailingStop *Point);
            if( (OrderProfit() > 0) && (OrderStopLoss() < dTrailingStop) )
            {
                bOrderModifyStatus = OrderModify(OrderTicket(),OrderOpenPrice(),dTrailingStop,
                    OrderTakeProfit(), 0, Green);

                } // if OrderProfit
            } // end of if OP_BUY

        } // end of if symbol
    } // end of OrderSelect for loop
```

The following MQL code is used to modify the stoploss of a sell order.

```
iTotalTrades = OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;

    if( ( ( OrderSymbol() == Symbol() ) ) && ( ( OrderMagicNumber() == iMagicNumber ) ) )
    {
        if( OrderType() == OP_BUY )
        {
            dTrailingStop = Ask - (iTrailingStop *Point);
            if( (OrderProfit() > 0) && (OrderStopLoss() < dTrailingStop) )
            {
                bOrderModifyStatus = OrderModify(OrderTicket(),OrderOpenPrice(),dTrailingStop,
                    OrderTakeProfit(), 0, Green);

                } // if OrderProfit
            } // end of if OP_BUY

            if( OrderType() == OP_SELL )
            {
                dTrailingStop = Bid + (iTrailingStop *Point);
                if if( (OrderProfit() > 0) && (OrderStopLoss() > dTrailingStop) )
                {
                    bOrderModifyStatus = OrderModify(OrderTicket(),OrderOpenPrice(),dTrailingStop,
                        OrderTakeProfit(), 0, Green);

                    } // end OrderModify
                } // end of if OP_SELL
            } // end of if symbol
        } // end of OrderSelect for loop
```

The following is the complete MQL code of the *fnTrailing* stop function.

```
void fnTrailingStop()
{
    string strErrorMessage;
    int iErrorNumber;
    bool bOrderModifyStatus;

    iTotalTrades = OrdersTotal();
    for(int pos=0;pos<iTotalTrades;pos++)
    {
        if(OrderSelect(pos,SELECT_BY_POS)==false)
            continue;

        if( (( OrderSymbol() == Symbol() ) ) && (( OrderMagicNumber() == iMagicNumber ) ) )
        {
            if( OrderType() == OP_BUY )
            {
                dTrailingStop = Ask - (iTrailingStop *Point);
                if( (OrderProfit() > 0) && (OrderStopLoss() < dTrailingStop) )
                {
                    bOrderModifyStatus = OrderModify(OrderTicket(),OrderOpenPrice(),dTrailingStop,
                                                        OrderTakeProfit(), 0, Green);
                    if( bOrderModifyStatus == false )
                    {
                        iErrorNumber = GetLastError();
                        strErrorMessage = ErrorDescription(iErrorNumber);
                        Print( "OrderModify Failed: ", strErrorMessage );
                        return;
                    } // end of if bOrderCloseStatus
                } // if OrderProfit
            } // end of if OP_BUY

            if( OrderType() == OP_SELL )
            {
                dTrailingStop = Bid + (iTrailingStop *Point);
                if( (OrderProfit() > 0) && ( ) )
                {
                    bOrderModifyStatus = OrderModify(OrderTicket(),OrderOpenPrice(),dTrailingStop,
                                                        OrderTakeProfit(), 0, Green);
                    if( (OrderProfit() > 0) && (OrderStopLoss() > dTrailingStop) )
                    {
                        iErrorNumber = GetLastError();
                        strErrorMessage = ErrorDescription(iErrorNumber);
                        Print( "OrderModify Failed: ", strErrorMessage );
                        return;
                    } // end of if bOrderCloseStatus
                } // end OrderModify
            } // end of if OP_SELL
        } // end of if sysmbol
    } // end of OrderSelect for loop
} // end fnTrailingStop
```

- Note that error handling implemented on the return value of the MQL function *OrderModify*.
- Note the return statement after the *OrderModify* function call. This is to obey the 5-second trade server rule.

Multiple Trades/Single Trade

One problem with the **iStarter** trading system is that it may open multiple trades based on the same signal. (These can be winning or losing trades).

Inspecting the open trade logic reveals the potential for this issue:

Open a buy position when the Ask price is higher than the 12-period exponential moving average and the 14-period RSI indicator is less than 40.

The following sequence of events may occur.

- The open trade logic evaluates to true.
- A buy trade is opened.
- The buy trade reaches its takeprofit or stoploss
- The open trade logic *still* evaluates to true. (It never changed).

One method to address this issue is to only allow another trade to be opened in the opposite direction. To accomplish this, the Expert Advisor will need to save the last opened trade type.

Declare a new integer variable named **iLastOpenTradeType**.

```
// Open Trade Information Variables
int iLastTradeType;
int iTotalTrades;
double dTotalTradeLots;
double dTotalTradeSwap;
int iTotalBuyTrades;
int iTotalSellTrades;
```

Add the **init** function to the Expert Advisor. Inside of the **init** function, set the **iLastOpenTradeType** variable to -1. (This will only be set to (-1) once - when the Expert Advisor is first attached to the chart.)

```
// The init function, only execute the first time the Expert Advisor
// is attached to a chart
int init()
{
    iLastTradeType=-1;
}

// The entry point of the Expert Advisor: the start function
int start()
{
    fnGetOpenTradeInfo();
    fnOpenBuyLogic();
    fnOpenSellLogic();
}
```

In the *fnOpenOrderInfo* function, set the *iLastOpenTradeType* variable to **OP_BUY** or **OP_SELL** if there is an open buy or sell trade.

```
iTotalTrades=OrdersTotal();
for(int pos=0;pos<iTotalTrades;pos++)
{
    if(OrderSelect(pos,SELECT_BY_POS)==false)
        continue;
    if( (( OrderSymbol() == Symbol() ) ) && (( OrderMagicNumber() == iMagicNumber ) ) )
    {

        iTotalSelectedTrades++;
        dTotalOpenProfit += OrderProfit();
        dTotalTradeSwap += OrderSwap();
        dTotalTradeLots += OrderLots();
        dTotalOpenProfit += OrderProfit();
        dTotalTradeSwap += OrderSwap();
        if( OrderType() == OP_BUY )
        {
            iTotalBuyTrades++;
            iLastTradeType = OP_BUY;

        }
        if( OrderType() == OP_SELL )
        {
            iTotalSellTrades++;
            iLastTradeType=OP_SELL;

        }

    }
}
```

Add logic to the *fnOpenBuyLogic* and *fnOpenSellLogic* functions to only set the trade signal if the *iLastOpenTradeType* variable is equal to -1 or the opposite trade type.

```
// This will set the variable iTradeSignal to iOpenBuySignal if the logic
evaluates to true.
void fnOpenBuyLogic()
{
    double dMovingAvg=0;
    dMovingAvg = iMA(NULL,NULL,12,0,MODE_EMA,PRICE_CLOSE,0);

    double dRelativeStrength =0;
    dRelativeStrength = iRSI(NULL,NULL,12,PRICE_CLOSE,0);

    if( (Ask > dMovingAvg) && (dRelativeStrength > 40.0) )
    {
        if( (iLastTradeType == OP_SELL) || (iLastTradeType == -1) )
            iTradeSignal = iOpenBuySignal;
    }
}

// The logic for opening a Sell order.
// This will set the variable iTradeSignal to iOpenSellSignal if the logic
evaluates to true.
void fnOpenSellLogic()
{
    int iLowestIndex = iLowest(NULL,NULL,MODE_LOW,18,0);
    double dLowestValue = Low[iLowestIndex];

    if( Bid < dLowestValue )
    {
        if( (iLastTradeType == OP_BUY) || (iLastTradeType == -1) )
            iTradeSignal = iOpenSellSignal;
    }
}
```

No Trade Signal to Close Position

Many traders do use an indicator-based trade signal to close a position. They simply use the stoploss and takeprofit limits to manage the open trade. This is easy to do with the **iStarter** Expert Advisor. Perhaps the simplest way to accomplish this is to comment out, or remove, the calls to the **fnCloseBuyLogic** and **fnCloseSellLogic** functions within the start function:

```
// fnCloseBuyLogic();  
// fnCloseSellLogic();
```

Controlling Trade Open

Many traders wish to add further criteria to the trade open logic. One clean method to do this is to create a new user-defined function called **fnOkToOpen**. The function **fnOkToOpen** implements logic and changes the value of the **iTradeSignal** variable if applicable. **fnOkToOpen** should be called before the function **fnOrder**.

The function **fnOkToOpen** can be used to limit the number of trades opened by the Expert Advisor.

```
void fnOkToOpen()  
{  
    if( iTotalSelectedTrades >= iMaxTrades )  
    {  
        if( iTradeSignal > 0 )  
            iTradeSignal = iNoSignal;  
    }  
}
```

fnOkToOpen can also be used to control the time when trades can be opened.

```
extern int iStartHour=7;  
extern int iEndHour=22;  
  
void fnOkToOpen()  
{  
    if( iTotalSelectedTrades >= iMaxTrades )  
    {  
        if( iTradeSignal > 0 )  
            iTradeSignal = iNoSignal;  
    }  
  
    if( (Hour() < iStartHour) || (Hour() > iEndHour) )  
    {  
        if( iTradeSignal > 0 )  
            iTradeSignal = iNoSignal;  
    }  
}
```

The **start** function should call the *fnOkToOpen* function before *fnOrder* function.

```
// The entry point of the Expert Advisor: the start function
int start()
{
    fnGetOpenTradeInfo();
    fnOpenBuyLogic();
    fnOpenSellLogic();
    fnCloseBuyLogic();
    fnCloseSellLogic();
    fnOkToOpen();
    fnOrder();

    return(0);
}
```

Changing the *iStarter* Expert Advisor

One of the main objectives of using the *iExpertAdvisor* method is to create an Expert Advisor that is easy to change.

To reiterate the behavior of the *iStarter* Expert Advisor:

- Collect information about open positions (*fnOpenOrderInfo*).
- Determine if the market is generation a signal to open a buy or sell position (*fnOpenBuyLogic* and *fnCloseBuyLogic*).
- Determine if the market is generation a signal to close a buy or sell position (*fnOpenSellLogic* and *fnCloseSellLogic*).
- Check additional open or close criteria: time of day; number of open trades, etc. (*fnOkToOpen*).
- Open or close an order if the trade signal is set (*fnOrder*).
- Manage an open position: apply a trailing stop, move a stop to breakeven, etc. (*fnManageTrades*).

The changes needed to implement new functionality usually falls into one of these functions.

- To make a change to the criteria used to open a Buy position, change *fnOpenBuyLogic*.
- To make a change to the criteria used to close a Buy position, change *fnCloseBuyLogic*.
- To make a change to the criteria used to open a Sell position, change *fnOpenSellLogic*.
- To make a change to the criteria used to close a Sell position, change *fnCloseSellLogic*.
- To make a change to the trailing stop, or to add more advanced stops such as a trending stop or a channel stop, change *fnManageTrades*.

Changes that require information about the status of the opens trades may require adding variables to the *fnOpenOrderInfo* function.

For example, to close a trade when its swap value has exceeded a threshold:

- Create a new variable: double *dTotalOpenSwap*
- Set the variable to zero before the start of the *for* loop in the function *fnOpenOrderInfo*.
- Within the for loop, use the MQL function *OrderSwap* to store the swap value:
dTotalOpenSwap += OrderSwap();

Some changes may require adding new user-defined functions. The function *fnOrder* should not require changes very often. However, a trader may wish to adjust the Lot size used to open or close an order before calling the *fnOrder* function. A new user-defined function can be created named *fnSetLotSize*. This function can be used to set the variable *dLot* based on money management, a winning or losing trade, etc.

The Order window

Order

Symbol: EURUSD, Euro vs US Dollar

Volume: 1.00

Stop Loss: 0.0000 Take Profit: 0.0000

Comment:

Type: Instant Execution

Instant Execution

1.2793 / 1.2795

Sell Buy

☐ Enable maximum deviation from quoted price

Maximum deviation: 0 pips

Appendix 1 - Glossary

Alert - an MQL function that send a text message to the Alert window.

Arguments- see parameters.

Array – a list of data types in memory. The data types may be string, double or integer.

Ask – the price at which broker/dealer is willing to sell a currency pair.

Bar – another name for candle or candlestick.

Bid – the price at which broker/dealer is willing to buy a currency pair.

Braces – { } Curly braces are used to define scope. They are used define the start and end of functions as well as the *if*, *for*, *while* and *switch* operations.

Bug – an inadvertent logically decision within a software program.

Buy (order) – to open a long position of a currency pair.

Candle– (Candlestick) a diagram that encapsulates the high, low, open and close price of a period.

Chart – a price chart, often with candlesticks.

Client – the requester of services within a client-server paradigm.

Close (price) – the price at which a currency pair closed at for a given time period.

Close (order) – to close a position, long or short, of a currency pair.

Comment – an MQL function that writes text to the chart.

Comments – text within an MQL program that is ignored by the compiler. Within // or /* */.

Compile – to convert human readable text into machine readable binary code.

Compile Time - time at which an Expert Advisor's MQL code is converted into executable object code.

Compiler – an application that compiles MQL source code.

Copyright – an MQL property that can be set within an MQL file.

Crossover – a crossover of line A and line B occurs when the value at time 0 of A is less than B and the value at time 1 of A is greater than B (or vice-versa).

Custom Indicator – an indicator written in MQL using the MetaEditor.

Custom Indicator Usage - see appendix 16.

Data Types – A portion of memory of a specific size used to hold data. MQL supports the following data types: integer, double, string, Boolean, datetime and array.

Debugging – the process of verifying the intended functionality of an Expert Advisor and adjusting logic to provide the intended functionality.

Declaration – a variable is declared when it is defined within an Expert Advisor. The declaration includes the data type of the variable, the name and optionally a default value. Example: *int i =0;*

Default – the behavior of an entity if no other behavior is specified.

DelInit - a predefined function of an Expert Advisor. The exit point of an Expert Advisor.

Digits – the number of digits after the decimal point for a currency pair. Available in MQL via the *Digits* function or the *MarketData* function.

DLL – a Dynamic Link Library. A portion of executable code that can be loaded while an application is running. The term DLL is specific to the Microsoft Windows operating system; however the concept is shared among other operating systems such as Unix and MacOS.

Else – a conditional operator supported by the MQL language. Always used with a corresponding *if* operator.

Entry Point – the location where a program begins execution. The **start** function is the entry point of an Expert Advisor.

Errors – numerical values returned from MQL functions indicating the requested functionality was not performed successfully.

Expert Advisor – an object file interpreted by the MetaTrader platform that can manage trading positions based on programmed logic.

Extern Variables – variable that are available to the user when starting an Expert Advisor. These variables can also be used in the Strategy Tester to optimize the Expert Advisor.

For – a looping operator supported by the MQL language.

Forex - Short for the foreign exchange market. A market where buyers and sellers conduct foreign exchange transactions.

Function – a portion of MQL code, usually defined for a specific purpose.

Function Parameters – see parameters.

Function Signature – the complete description of a function: return value; name; parameter types and names.

Global Scope – the area of MQL code outside of all/any curly braces.

High (price) – the highest price a currency pair reached during a given time period.

If - a conditional operator supported by the MQL language. May be used with a corresponding *else* operator.

Include – a keyword used to include the contents of another MQL file.

Indentation – a method of aligning MQL code to improve readability. Usually aligned by curly braces.

Init – a predefined function of an Expert Advisor. The entry point the first time the Expert Advisor is run.

Keywords – specially defined words of the MQL language.

Library – a collection of object code, usually one or more functions.

Looping – a construct used in programming to perform similar logic more than once.

Low (price) – the lowest price a currency pair reached during a given time period.

Low/Lowest – the low price of a bar or group of bars.

Magic Number - the magic number is a unique number that is assigned to an order when it is opened. For example, when an Expert Advisor opens an order on the **EURUSD**, it can assign a magic number of 3000. Later when the Expert Advisor requests information about all of the open orders for the account, it recognizes the orders it has opened by their magic number - any order whose magic number is 3000. The magic number allows an Expert Advisor a method to tag and identify orders.

Market Order – an order that is immediately routed to the market and is executed at the current price.

MetaEditor – the editor and compiler of the MetaTrader platform. Use to write and build Expert Advisor, scripts, and custom indicators.

MetaTrader - a FOREX trading platform that supports automatic trading using Expert Advisors.

MQL – Meta Query Language. A programming language used to build Expert Advisor, scripts, and custom indicators.

NULL – an unset value, usually zero.

Object Code – non human readable data that is interpreted by an application or an operating system.

Objects – any graphical element on a price chart (lines, arrows, etc.).

OP_BUYLIMIT – a pending order that opens a long position when the price breaks below a level.

OP_BUYSTOP - a pending order that opens a long position when the price breaks above a level.

OP_SELLLIMIT - a pending order that opens a short position when the price breaks above a level.

OP_SELLSTOP - a pending order that opens a short position when the price breaks below a level.

Open (price) – the price at which a currency pair opened at for a given time period.

Open (order) – to create a new position, long or short, of a currency pair.

Optional Parameters – parameters of a function that are not required to be defined by the caller of the function. If the parameter is not supplied, it assumes the values defined in the function signature.

Example: `bool OrderSelect(int index, int select, int pool=MODE_TRADES)`. The parameter will assume the value `MODE_TRADES` if the parameter is not provided when the function is called.

Order Type – market order types are `OP_BUY` and `OP_SELL`. Pending order types are: `OP_BUYLIMIT`, `OP_BUYSTOP`, `OP_SELLLIMIT`, and `OP_SELLSTOP`.

Parameters – the input to a user-defined or MQL function.

Pending Order – an order set on the broker's server to be converted to a market order based on price specific movement.

Point – or Pip, stands for Percentage In Point. It is the smallest price unit of currency pair. Some currency pairs are quoted to the fourth decimal point (EURUSD) while others are quoted to the second decimal point (JPYUSD). Available in MQL via the **Point** function or the **MarketData** function.

Predefined Variables –variables defined by the MQL language.

Preferred Closing Price – the preferred closing price when closing an open order. The Preferred closing price for an `OP_BUY` position is the Bid price. The Preferred closing price for an `OP_SELL` position is the Ask price.

Price Chart – a chart showing the prices of a currency pair. The chart types supported by the MetaTrader platform are **Bar**, **Candlestick** and **Line**.

Reserved Words – textual words used by the MQL language for special purposes.

Return – the MQL reserved word **return** is the exit point of a function.

Return Value – a function may use the MQL reserved word **return** to return a value. Example: `return(0);`

Run Time – the time at which an application, or Expert Advisor, is executing, or running its code.

Scope – an area of MQL code – usually defined by curly braces.

Script – compiled MQL code that executes once when selected.

Sell (order) - to open a short position of a currency pair.

SendEmail – an MQL function used to send email. The MetaTrader platform must be configured to send email.

Series – an array or list of data. The built-in MQL series are: High, Low, Open, and Close.

Server – the provider of services within a client-server paradigm.

Shift – The bar or candle that is currently forming (furthest to the right on a candlestick chart) is numbered zero, each bar to the left increases in number. A series, or array, of Indicator values also follow this concept of number 0 starting at the far right with each number increasing to the left. The

Shift of an indicator value is the index in the array of the value, where shift=0 is the latest value and shift=1 is the value from the last period.

Slippage – the number of points the entry price may move and still allow the order to be opened.

Spread - the distance, usually in pips, between the **Bid** and **Ask** price.

Start - a predefined function of an Expert Advisor. The entry point after the first time the Expert Advisor is run.

Stoploss – a value set with an order that instructs the trade server to close the order when a price has reached the stoploss limit. An Expert Advisor does not need to be running for an order to close based on its stoploss value.

Strategy Tester – The MetaTrader platform back tester program. To view: View->Strategy Tester.

Structured Programming Language – a programming language that supports user-defined variables and functions.

Symbol – the name of a currency pair. Example: EURUSD.

Syntax – the grammatical rules of a programming language.

Takeprofit - a value set with an order that instructs the trade server to close the order when a price has reached the takeprofit limit. An Expert Advisor does not need to be running for an order to close based on its takeprofit value.

Template – (1) a skelton of MQL code; (2) a collection of platform user settings.

Terminal – the MetaTrader platform executable application.

Testing – the process of verifying the performance of an Expert Advisor using historical data. Testing implies that debugging has been performed and that the Expert Advisor is executing the intended logic.

Tick – the price of a currency pair at an exact instance in time.

Trade Server – the broker's server that open and closes currency positions.

Trade Set – the collection of trades managed by an Expert Advisor. May be all trades for an account, or may be based on the trade's currency symbol and/or magic number.

TradeSignal – a variable defined in the iExpertAdvisor method to control opening and closing trades.

Trailing Stop – a stoploss value that changes as the market price changes.

Variables – defined data types, used to hold user data.

While – a looping operator supported by the MQL language.

Appendix 2 – Reserved Words

The identifiers listed below are fixed reserved words. A certain action is assigned to each of them, and they cannot be used for other purposes:

Data types	Memory classes	Operators	Other
bool	extern	break	false
color	static	case	true
datetime		continue	
double		default	
int		else	
string		for	
void		if	
		return	
		switch	
		while	

Appendix 3 – Color Constants

Color type supported color constants.

Black	DarkGreen	DarkSlateGray	Olive	Green	Teal	Navy	Purple
Maroon	Indigo	MidnightBlue	DarkBlue	DarkOliveGreen	SaddleBrown	ForestGreen	OliveDrab
SeaGreen	DarkGoldenrod	DarkSlateBlue	Sienna	MediumBlue	Brown	DarkTurquoise	DimGray
LightSeaGreen	DarkViolet	FireBrick	MediumVioletRed	MediumSeaGreen	Chocolate	Crimson	SteelBlue
Goldenrod	MediumSpringGreen	LawnGreen	CadetBlue	DarkOrchid	YellowGreen	LimeGreen	OrangeRed
DarkOrange	Orange	Gold	Yellow	Chartreuse	Lime	SpringGreen	Aqua
DeepSkyBlue	Blue	Magenta	Red	Gray	SlateGray	Peru	BlueViolet
LightSlateGray	DeepPink	MediumTurquoise	DodgerBlue	Turquoise	RoyalBlue	SlateBlue	DarkKhaki
IndianRed	MediumOrchid	GreenYellow	MediumAquamarine	DarkSeaGreen	Tomato	RosyBrown	Orchid
MediumPurple	PaleVioletRed	Coral	CornflowerBlue	DarkGray	SandyBrown	MediumSlateBlue	Tan
DarkSalmon	BurlyWood	HotPink	Salmon	Violet	LightCoral	SkyBlue	LightSalmon
Plum	Khaki	LightGreen	Aquamarine	Silver	LightSkyBlue	LightSteelBlue	LightBlue
PaleGreen	Thistle	PowderBlue	PaleGoldenrod	PaleTurquoise	LightGray	Wheat	NavajoWhite
Moccasin	LightPink	Gainsboro	PeachPuff	Pink	Bisque	LightGoldenrod	BlanchedAlmond
LemonChiffon	Beige	AntiqueWhite	PapayaWhip	Cornsilk	LightYellow	LightCyan	Linen
Lavender	MistyRose	OldLace	WhiteSmoke	Seashell	Ivory	Honeydew	AliceBlue
LavenderBlush	MintCream	Snow	White				

Appendix 4 – Operation Precedence

Each group of operations in the table has the same priority. The higher is the priority, the higher is the position of the group in the table. The precedence rules determine the grouping of operations and operands.

()	Function call	From left to right
[]	Referencing to an array element	
!	Logical negation	From right to left
-	Sign changing operation	
++	Increment	
--	Decrement	
~	Bitwise negation (complement)	
&	Bitwise operation AND	From left to right
	Bitwise operation OR	
^	Bitwise operation exclusive OR	
<<	Left shift	
>>	Right shift	
*	Multiplication	From left to right
/	Division	
%	Module division	
+	Addition	From left to right
-	Subtraction	
<	Less than	From left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
==	Equal	
!=	Not equal	
	Logical OR	From left to right
&&	Logical AND	From left to right
=	Assignment	From right to left
+=	Assignment addition	
-=	Assignment subtraction	
*=	Assignment multiplication	
/=	Assignment division	
%=	Assignment module	
>>=	Assignment right shift	
<<=	Assignment left shift	
&=	Assignment bitwise AND	
=	Assignment bitwise OR	
^=	Assignment exclusive OR	
,	Comma	From left to right

Parentheses that have higher priority are applied to change the execution order of the operations.

Appendix 5 – Arithmetic Operators

The MQL Arithmetic Operators

Operator Symbol	Operation	Example
+	Addition	$10 + 3 = 13$
-	Subtraction	$10 - 3 = 7$
*	Multiplication	$10 * 3 = 30$
/	Division	$10/3 = 3.33333$
%	Modulo (remainder)	Remainder of $(10/3) = 1$

Appendix 6 – Error Codes

Error codes returned from trade server.

Constant	Value	Description
ERR_NO_ERROR	0	No error returned.
ERR_NO_RESULT	1	No error returned, but the result is unknown.
ERR_COMMON_ERROR	2	Common error.
ERR_INVALID_TRADE_PARAMETERS	3	Invalid trade parameters.
ERR_SERVER_BUSY	4	Trade server is busy.
ERR_OLD_VERSION	5	Old version of the client terminal.
ERR_NO_CONNECTION	6	No connection with trade server.
ERR_NOT_ENOUGH_RIGHTS	7	Not enough rights.
ERR_TOO_FREQUENT_REQUESTS	8	Too frequent requests.
ERR_MALFUNCTIONAL_TRADE	9	Malfunctional trade operation.
ERR_ACCOUNT_DISABLED	64	Account disabled.
ERR_INVALID_ACCOUNT	65	Invalid account.
ERR_TRADE_TIMEOUT	128	Trade timeout.
ERR_INVALID_PRICE	129	Invalid price.
ERR_INVALID_STOPS	130	Invalid stops.
ERR_INVALID_TRADE_VOLUME	131	Invalid trade volume.
ERR_MARKET_CLOSED	132	Market is closed.
ERR_TRADE_DISABLED	133	Trade is disabled.
ERR_NOT_ENOUGH_MONEY	134	Not enough money.
ERR_PRICE_CHANGED	135	Price changed.
ERR_OFF_QUOTES	136	Off quotes.
ERR_BROKER_BUSY	137	Broker is busy.
ERR_REQUOTE	138	Requote.
ERR_ORDER_LOCKED	139	Order is locked.
ERR_LONG_POSITIONS_ONLY_ALLOWED	140	Long positions only allowed.
ERR_TOO_MANY_REQUESTS	141	Too many requests.
ERR_TRADE_MODIFY_DENIED	145	Modification denied because order too close to market.
ERR_TRADE_CONTEXT_BUSY	146	Trade context is busy.
ERR_TRADE_EXPIRATION_DENIED	147	Expirations are denied by broker.
ERR_TRADE_TOO_MANY_ORDERS	148	The amount of open and pending orders has reached the limit set by the broker.

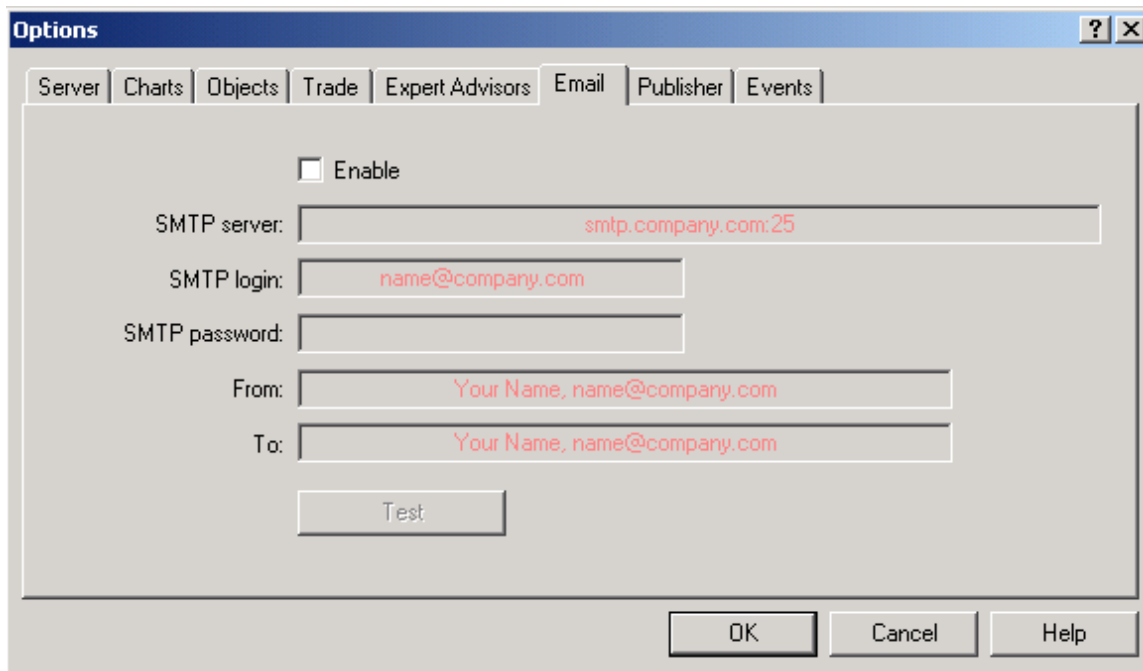
MQL4 run time error codes

Constant	Value	Description
ERR_NO_MQLERROR	4000	No error.
ERR_WRONG_FUNCTION_POINTER	4001	Wrong function pointer.
ERR_ARRAY_INDEX_OUT_OF_RANGE	4002	Array index is out of range.
ERR_NO_MEMORY_FOR_CALL_STACK	4003	No memory for function call stack.
ERR_RECURSIVE_STACK_OVERFLOW	4004	Recursive stack overflow.
ERR_NOT_ENOUGH_STACK_FOR_PARAM	4005	Not enough stack for parameter.
ERR_NO_MEMORY_FOR_PARAM_STRING	4006	No memory for parameter string.
ERR_NO_MEMORY_FOR_TEMP_STRING	4007	No memory for temp string.
ERR_NOT_INITIALIZED_STRING	4008	Not initialized string.
ERR_NOT_INITIALIZED_ARRAYSTRING	4009	Not initialized string in array.
ERR_NO_MEMORY_FOR_ARRAYSTRING	4010	No memory for array string.
ERR_TOO_LONG_STRING	4011	Too long string.
ERR_REMAINDER_FROM_ZERO_DIVIDE	4012	Remainder from zero divide.
ERR_ZERO_DIVIDE	4013	Zero divide.
ERR_UNKNOWN_COMMAND	4014	Unknown command.
ERR_WRONG_JUMP	4015	Wrong jump (never generated error).
ERR_NOT_INITIALIZED_ARRAY	4016	Not initialized array.
ERR_DLL_CALLS_NOT_ALLOWED	4017	DLL calls are not allowed.
ERR_CANNOT_LOAD_LIBRARY	4018	Cannot load library.
ERR_CANNOT_CALL_FUNCTION	4019	Cannot call function.
ERR_EXTERNAL_CALLS_NOT_ALLOWED	4020	Expert function calls are not allowed.
ERR_NO_MEMORY_FOR_RETURNED_STR	4021	Not enough memory for temp string returned from function.
ERR_SYSTEM_BUSY	4022	System is busy (never generated error).
ERR_INVALID_FUNCTION_PARAMSCNT	4050	Invalid function parameters count.
ERR_INVALID_FUNCTION_PARAMVALUE	4051	Invalid function parameter value.
ERR_STRING_FUNCTION_INTERNAL	4052	String function internal error.
ERR_SOME_ARRAY_ERROR	4053	Some array error.
ERR_INCORRECT_SERIESARRAY_USING	4054	Incorrect series array using.
ERR_CUSTOM_INDICATOR_ERROR	4055	Custom indicator error.
ERR_INCOMPATIBLE_ARRAYS	4056	Arrays are incompatible.

ERR_GLOBAL_VARIABLES_PROCESSING	4057	Global variables processing error.
ERR_GLOBAL_VARIABLE_NOT_FOUND	4058	Global variable not found.
ERR_FUNC_NOT_ALLOWED_IN_TESTING	4059	Function is not allowed in testing mode.
ERR_FUNCTION_NOT_CONFIRMED	4060	Function is not confirmed.
ERR_SEND_MAIL_ERROR	4061	Send mail error.
ERR_STRING_PARAMETER_EXPECTED	4062	String parameter expected.
ERR_INTEGER_PARAMETER_EXPECTED	4063	Integer parameter expected.
ERR_DOUBLE_PARAMETER_EXPECTED	4064	Double parameter expected.
ERR_ARRAY_AS_PARAMETER_EXPECTED	4065	Array as parameter expected.
ERR_HISTORY_WILL_UPDATED	4066	Requested history data in updating state.
ERR_TRADE_ERROR	4067	Some error in trading function.
ERR_END_OF_FILE	4099	End of file.
ERR_SOME_FILE_ERROR	4100	Some file error.
ERR_WRONG_FILE_NAME	4101	Wrong file name.
ERR_TOO_MANY_OPENED_FILES	4102	Too many opened files.
ERR_CANNOT_OPEN_FILE	4103	Cannot open file.
ERR_INCOMPATIBLE_FILEACCESS	4104	Incompatible access to a file.
ERR_NO_ORDER_SELECTED	4105	No order selected.
ERR_UNKNOWN_SYMBOL	4106	Unknown symbol.
ERR_INVALID_PRICE_PARAM	4107	Invalid price.
ERR_INVALID_TICKET	4108	Invalid ticket.
ERR_TRADE_NOT_ALLOWED	4109	Trade is not allowed. Enable checkbox "Allow live trading" in the expert properties.
ERR_LONGS_NOT_ALLOWED	4110	Longs are not allowed. Check the expert properties.
ERR_SHORTS_NOT_ALLOWED	4111	Shorts are not allowed. Check the expert properties.
ERR_OBJECT_ALREADY_EXISTS	4200	Object exists already.
ERR_UNKNOWN_OBJECT_PROPERTY	4201	Unknown object property.
ERR_OBJECT_DOES_NOT_EXIST	4202	Object does not exist.
ERR_UNKNOWN_OBJECT_TYPE	4203	Unknown object type.
ERR_NO_OBJECT_NAME	4204	No object name.
ERR_OBJECT_COORDINATES_ERROR	4205	Object coordinates error.
ERR_NO_SPECIFIED_SUBWINDOW	4206	No specified subwindow.
ERR_SOME_OBJECT_ERROR	4207	Some error in object function.

Appendix 7 – Configuring Email

To enable email, from the MetaTrader Terminal, navigate to the **Tools->Options** menu, then select the **Email** tab to view the following window.

The image shows a screenshot of the 'Options' dialog box in MetaTrader, with the 'Email' tab selected. The dialog has a title bar with a question mark and a close button. Below the title bar is a tabbed interface with tabs for 'Server', 'Charts', 'Objects', 'Trade', 'Expert Advisors', 'Email', 'Publisher', and 'Events'. The 'Email' tab is active. Inside the tab, there is a checkbox labeled 'Enable' which is currently unchecked. Below this are several text input fields: 'SMTP server:' with the placeholder text 'smtp.company.com:25', 'SMTP login:' with the placeholder text 'name@company.com', 'SMTP password:' (empty), 'From:' with the placeholder text 'Your Name, name@company.com', and 'To:' with the placeholder text 'Your Name, name@company.com'. Below these fields is a 'Test' button. At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Help'.

Check the **Enable** checkbox and fill in the required information. If you do not know the name of your SMTP server, request this information from your ISP (Internet Server Provider). After entering the information, select **OK** to enable emailing.

The following information is from the MetaTrader documentation:

Email Tab:

In this tab, the electronic mailbox is set up. Later on, these settings will be used to send message by the expert advisor command or by a triggered alert. To start setting up of email, the "Enable" must be enabled and the following fields must be filled out:

- **SMTP server** — address of the SMTP server used and port. The given server will be used for sending messages. The entry must have the following format "[internet address-server address]: [port number]". For example, "192.168.0.1:443", where "192.168.0.1" is the server address, and "443" is the port number;

Copyright© *iExpertAdvisor*, LLC All Rights Reserved.

- **SMTP login** — login to authorize at the mail server;
- **SMTP password** — authorization password;
- **From** — electronic mail address from which messages will be sent;
- **To** — electronic mail address to which messages will be sent.

Attention: Only one email address may be specified for either of fields "From" and "To". Several emails given with or without separators will not be accepted.

The "Test" sends a test message using the settings specified to test their workability. If it has been tested successfully, the "OK" button must be pressed to apply these settings. If test is unsuccessful, it is recommended to check all settings and resend the test message.

Appendix 8 – Preferred Prices

Although the MQL *OrderSend* and *OrderClose* functions accept a *Slippage* parameter these functions will sometimes fail if the price is not set to the following values.

Function	Preferred Price
Open Buy	Ask
Open Sell	Bid
Close Buy	Bid
Close Sell	Ask

Appendix 9 – MarketInfo Function

The MQL function **MarketInfo** is generally used to collect information about currencies pairs. The Expert Advisor does not need to be attached to the symbol's chart to retrieve this information.

For example, the MQL predefined variable **Ask** contains the value of the **Ask** price for the currency to which the Expert Advisor is attached. If the Expert Advisor is attached to a EURUSD chart, the value of **Ask** is the **Ask** price for the EURUSD currency pair. The value of the **Ask** price of the GBPUSD currency can be retrieved using the **MarketInfo** function:

```
double dGbpUsdAsk = MarketInfo("GBPUSD",MODE_ASK);
```

The signature of the **MarketInfo** function is:

```
double MarketInfo(          string symbol, int type)
```

The supported types are:

Constant	Value	Description
MODE_LOW	1	Low day price.
MODE_HIGH	2	High day price.
MODE_TIME	5	The last incoming tick time (last known server time).
MODE_BID	9	Last incoming bid price. For the current symbol, it is stored in the predefined variable Bid
MODE_ASK	10	Last incoming ask price. For the current symbol, it is stored in the predefined variable Ask
MODE_POINT	11	Point size in the quote currency. For the current symbol, it is stored in the predefined variable Point
MODE_DIGITS	12	Count of digits after decimal point in the symbol prices. For the current symbol, it is stored in the predefined variable Digits
MODE_SPREAD	13	Spread value in points.
MODE_STOPLEVEL	14	Stop level in points.
MODE_LOTSIZE	15	Lot size in the base currency.
MODE_TICKVALUE	16	Tick value in the deposit currency.
MODE_TICKSIZE	17	Tick size in the quote currency.
MODE_SWAPLONG	18	Swap of the long position.
MODE_SWAPSHORT	19	Swap of the short position.
MODE_STARTING	20	Market starting date (usually used for futures).

MODE_EXPIRATION	21	Market expiration date (usually used for futures).
MODE_TRADEALLOWED	22	Trade is allowed for the symbol.
MODE_MINLOT	23	Minimum permitted amount of a lot.
MODE_LOTSTEP	24	Step for changing lots.
MODE_MAXLOT	25	Maximum permitted amount of a lot.
MODE_SWAPTYPE	26	Swap calculation method. 0 - in points; 1 - in the symbol base currency; 2 - by interest; 3 - in the margin currency.
MODE_PROFITCALCMODE	27	Profit calculation mode. 0 - Forex; 1 - CFD; 2 - Futures.
MODE_MARGINCALCMODE	28	Margin calculation mode. 0 - Forex; 1 - CFD; 2 - Futures; 3 - CFD for indices.
MODE_MARGININIT	29	Initial margin requirements for 1 lot.
MODE_MARGINMAINTENANCE	30	Margin to maintain open positions calculated for 1 lot.
MODE_MARGINHEDGED	31	Hedged margin calculated for 1 lot.
MODE_MARGINREQUIRED	32	Free margin required to open 1 lot for buying.
MODE_FREEZELEVEL	33	Order freeze level in points. If the execution price lies within the range defined by the freeze level, the order cannot be modified, cancelled or closed.

Appendix 10 – Pending Order Definitions

Pending orders are opened using the **OrderSend** function. They are deleted using the **OrderDelete** function.

Note: Most Forex brokers do **not** support pending orders in the same fashion as pending orders are supported for equity positions. Typically a pending order is converted to market order when their criterion has been met. The market order is entered in the same fashion as a regular market order: the execute price is the market price plus or minus the slippage. *A pending order does not guarantee the execution price.*

Order Type	Definition
OP_BUYLIMIT	A pending order that opens a long position when the price breaks below a level.
OP_BUYSTOP	A pending order that opens a long position when the price breaks above a level.
OP_SELLLIMIT	A pending order that opens a short position when the price breaks above a level.
OP_SELLSTOP	A pending order that opens a short position when the price breaks below a level.

Appendix 11 - UninitializeReason

The MQL function **UninitializeReason** is used to determine the reason an Expert Advisor is exiting. It is most often placed in the **deinit** function.

The signature of the function is:

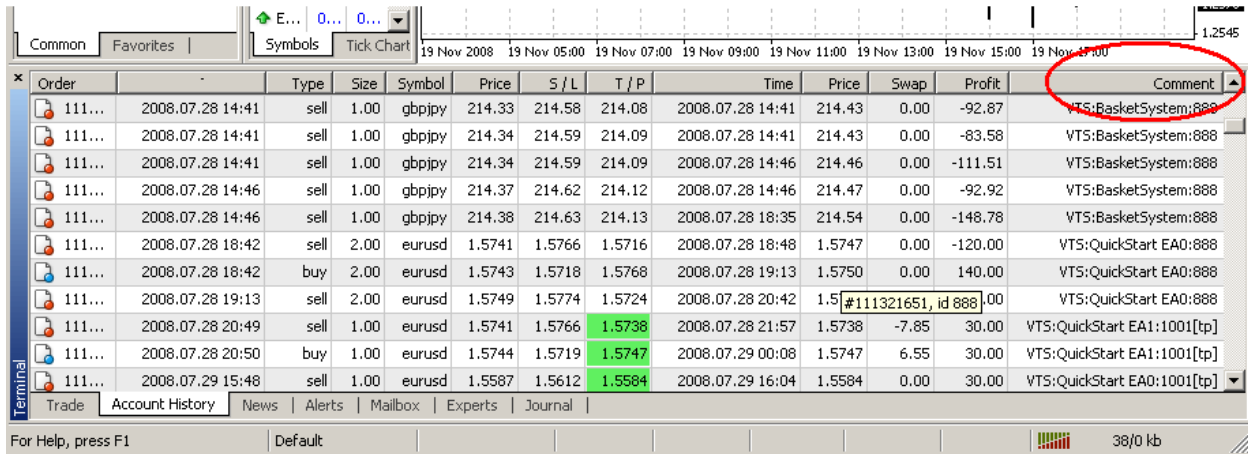
```
int UninitializeReason()
```

The reason codes returned. It can be any one of the following values:

Constant	Value	Description
	0	Script finished its execution independently.
REASON_REMOVE	1	Expert removed from chart.
REASON_RECOMPILE	2	Expert recompiled.
REASON_CHARTCHANGE	3	symbol or timeframe changed on the chart.
REASON_CHARTCLOSE	4	Chart closed.
REASON_PARAMETERS	5	Inputs parameters was changed by user.
REASON_ACCOUNT	6	Other account activated.

Appendix 12 – Using the Comment Field

The MQL function **OrderSend** provides a parameter to write a string into an order when it is created. The value of this string can be viewed from the Terminal's **Trade** or **Account History** tabs, under the **Comment** column. It can also be retrieved from an open order using the MQL function **OrderComment**.



Order		Type	Size	Symbol	Price	S / L	T / P	Time	Price	Swap	Profit	Comment
111...	2008.07.28 14:41	sell	1.00	gbpjpy	214.33	214.58	214.08	2008.07.28 14:41	214.43	0.00	-92.87	VTS:BasketSystem:888
111...	2008.07.28 14:41	sell	1.00	gbpjpy	214.34	214.59	214.09	2008.07.28 14:41	214.43	0.00	-83.58	VTS:BasketSystem:888
111...	2008.07.28 14:41	sell	1.00	gbpjpy	214.34	214.59	214.09	2008.07.28 14:46	214.46	0.00	-111.51	VTS:BasketSystem:888
111...	2008.07.28 14:46	sell	1.00	gbpjpy	214.37	214.62	214.12	2008.07.28 14:46	214.47	0.00	-92.92	VTS:BasketSystem:888
111...	2008.07.28 14:46	sell	1.00	gbpjpy	214.38	214.63	214.13	2008.07.28 18:35	214.54	0.00	-148.78	VTS:BasketSystem:888
111...	2008.07.28 18:42	sell	2.00	eurusd	1.5741	1.5766	1.5716	2008.07.28 18:48	1.5747	0.00	-120.00	VTS:QuickStart EA0:888
111...	2008.07.28 18:42	buy	2.00	eurusd	1.5743	1.5718	1.5768	2008.07.28 19:13	1.5750	0.00	140.00	VTS:QuickStart EA0:888
111...	2008.07.28 19:13	sell	2.00	eurusd	1.5749	1.5774	1.5724	2008.07.28 20:42	1.5747	0.00	30.00	VTS:QuickStart EA0:888
111...	2008.07.28 20:49	sell	1.00	eurusd	1.5741	1.5766	1.5738	2008.07.28 21:57	1.5738	-7.85	30.00	VTS:QuickStart EA1:1001[tp]
111...	2008.07.28 20:50	buy	1.00	eurusd	1.5744	1.5719	1.5747	2008.07.29 00:08	1.5747	6.55	30.00	VTS:QuickStart EA1:1001[tp]
111...	2008.07.29 15:48	sell	1.00	eurusd	1.5587	1.5612	1.5584	2008.07.29 16:04	1.5584	0.00	30.00	VTS:QuickStart EA0:1001[tp]

The comment value can also be used to manage open trades. For example, information about the Expert Advisor, the name, version, etc., can be stored in the comment field and later used by the Expert Advisor to make decisions. The MQL documentation clearly states the value of the comment field may be changed by the server. Experience has shown that the tradeserver modifies orders when they are closed by a takeprofit or a stoploss.

Appendix 13 – Using the *switch* Operator

The *switch* operator is a conditional operator (like the *if* operator). The switch operator is generally used when an *if* statement becomes too complex and the variables being tested are characters.

- The *switch* operator is used along with the reserved words *case*, *default* and *break*.
- The *switch* operator accepts a single parameter. This parameter is the variable of the conditional test.
- The *case* operator accepts a single parameter. This parameter is the value of the variable that results in a positive (true) condition.
- Multiple *case* operators are used within a single *switch* statement. Each *case* operator handles a specific value.
- The *break* operator is used to exit from the *switch* statement.
- The *default* operator is used to handle any value that is not handled by a specific *case* statement.

```
int character='a';

switch( character )
{
    case 'a':
        Print("The value of character is a");
        break;

    case 'b':
        Print("The value of character is b");
        break;

    case 'c':
        Print("The value of character is c");
        break;

    default:
        Print("This value of character is not handled:", character);
        break;
}
```

Note: If the *break* statement is omitted from a *case*, the *default* case will execute along with the specific case.

Appendix 14 – Crossovers and Bouncing Values

A crossover is a classic test condition used by the technical analyst. The statement usually reads something like “*open a long position when the fast moving average crosses up through the slow moving average*”.

Four values are required to test for a crossover condition: two values from the current period and two values from the last period. (The period can be any time frame from 1 minute to 1 week).

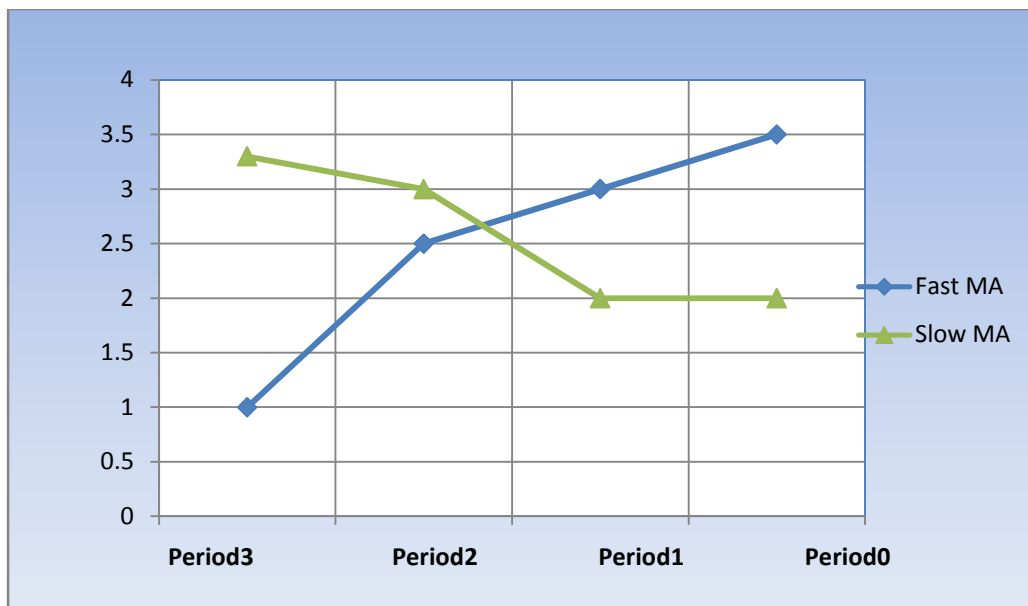
In the case of the moving averages, the values required are:

- Last period's fast moving average
- Last period's slow moving average
- This period's fast moving average
- This period's slow moving average

A crossover will have occurred if Last period's fast moving average was less than Last period's slow moving average; and This period's fast moving average is greater than This period's slow moving average.

The following chart illustrates the blue line (fast moving average) crossing up and through the green line (slow moving average). Notice the values:

- Period 2 Fast Moving Average = 2.5
- Period 2 Slow Moving Average = 3.0
- Period 1 Fast Moving Average = 3.0
- Period 1 Slow Moving Average = 2.0



The MQL series, such as **High**, **Low**, **Open**, and **Close** can be thought of as a list of values. Likewise, all of the MQL indicators can also be thought of as lists of values where the shift parameter is the index into the list. The following table lists some sample data values for the **Close** series and the fast and slow moving average indicators.

Shift	Index	Close	Fast MA	Slow MA
0	0	1.3040	1.3010	1.3060
1	1	1.3050	1.3020	1.3050
2	2	1.3060	1.3030	1.3040
3	3	1.3050	1.3040	1.3030
4	4	1.2040	1.2050	1.2020
5	5	1.3030	1.3060	1.3010

The signature of the **iMA** MQL function is:

```
double iMA( string symbol, int timeframe, int period, int ma_shift, int
ma_method, int applied_price, int shift)
```

Assume the fast period is 10 and the slow period is 20, the following MQL code stores the values of the moving averages need to test for a crossover. Note the shift parameter used to access the value.

```
double dFastMaPeriodTwo    = iMA(NULL, 0, 10, 0, MODE_SMA, PRICE_CLOSE, 2);
double dSlowMaPeriodTwo    = iMA(NULL, 0, 20, 0, MODE_SMA, PRICE_CLOSE, 2);
double dFastMaPeriodThree  = iMA(NULL, 0, 10, 0, MODE_SMA, PRICE_CLOSE, 3);
double dSlowMaPeriodThree  = iMA(NULL, 0, 20, 0, MODE_SMA, PRICE_CLOSE, 3);
```

The values of these variables (using the chart above and the shift parameter) are:

- dFastMaPeriodTwo = 1.3030
- dSlowMaPeriodTwo = 1.3040
- dFastMaPeriodThree = 1.3040
- dSlowMaPeriodThree = 1.303

The MQL code to test for the crossover is:

```
if( (dFastMaPeriodTwo < dSlowMaPeriodTwo) && (dFastMaPeriodThree > dSlowMaPeriodThree) )
{
    iTradeSignal = iOpenBuySignal;
}
```

This *if* statement will evaluate to *true* because:

1.3030 is less than 1.3040 AND 1.3040 is greater than 1.3030

This logic will work fine as long as the values being tested for the crossover are stable. If any of the values used for the crossover test are requested from the currently forming candle (shift=0 or index=0) then the result of conditional test may bounce. Because the candle is still forming, the values may change with each incoming tick. The cross may occur for just a second, but then quickly disappear. The human eye may not see this momentary crossover on a price chart, but an Expert Advisor will interpret it as a crossover – because the crossover conditional test *will* evaluate to true.

One simple method to avoid this problem entirely is to only test candles that have formed. That is, do not use the currently forming candle (shift=0 or index=0).

It is not always possible to avoid testing the currently forming candle. There are two common advanced methods:

Mandate that the crossover occurs *plus* some value. In this MQL code, the Fast Moving Average must at least move five points higher than the Slow Moving Average.

```
double dBuffer = 5 * Point;

if( (dFastMaPeriodTwo < dSlowMaPeriodTwo) && (dFastMaPeriodThree > (dSlowMaPeriodThree+dBuffer)))
{
    iTradeSignal = iOpenBuySignal;
}
```

Add a counter and only validate the crossover when the counter exceeds a set amount.

```
static int dBounceCount;

if( (dFastMaPeriodTwo < dSlowMaPeriodTwo) && (dFastMaPeriodThree > dSlowMaPeriodThree) )
{
    dBounceCount++;
}

if(dBounceCount > 10 )
{
    iTradeSignal = iOpenBuySignal;
}
```

Note: The bouncing problem can occur when using any value within a conditional test, not just crossovers.

Appendix 15 – Logging Information

There are two basic methods for getting information *out* of an Expert Advisor. The first is the MQL function *Print* and the second is the MQL function *Comment*.

The MQL function *Print* writes information to a log file and it also appears in the **Experts** tab of the Terminal and the **Journal** tab of the strategy tester.

The *Print* function is able to easily write any data type except for arrays. (An array must be written by addressing each index.)

```
int iInteger=9;
double dDouble = 10.1;
string strString = "hello";

Print( "iInteger=", iInteger, " dDouble=", dDouble, " strString=", strString );
```

There are special MQL functions for converting time data types into readable strings.

```
datetime iNow;
string strNow;

iNow = TimeCurrent();
strNow = TimeToStr(iNow);

Print( "iNow=", iNow, " strNow=", strNow );
```

Note: The *Print* function is the primary method used to debug MQL code.

The **Comment** function is used similarly to the **Print** function, however its data is written to the top left corner of price chart to which the Expert Advisor is attached.

Each call to the **Comment** function clears the previous information that was on the chart. In order to add to the data on the price chart, the data must be managed by the Expert Advisor. This is usually done by adding information to a single string variable throughout the Expert Advisor using the MQL function **StringConcatenate**. The **Comment** function is then called once before exiting the Expert Advisor.

```
string strComment;  
  
strComment = StringConcatenate(strComment, "Just called fnOpenTradeInfo" );  
strComment = StringConcatenate(strComment, "Just called fnOrder" );  
  
Comment(strComment);
```

The **Comment** function may also be useful for debugging. It is very useful for displaying the current state of the Expert Advisor (the value of the trade signal, the number trades it is managing, etc.).

Appendix 16 – The *iCustom* MQL Function

The information for the *iCustom* function is in the following table.

```
double iCustom(string symbol, int timeframe, string name, ..., int mode, int shift)
```

Calculates the specified custom indicator and returns its value. The custom indicator must be compiled (*.EX4 file) and be in the `terminal_directory\experts\indicators` directory.

Parameters:

symbol	- Symbol the data of which should be used to calculate indicator. NULL means current symbol.
timeframe	- Timeframe. It can be any of Timeframe enumeration values. 0 means the current chart timeframe.
name	- Custom indicator compiled program name.
...	- Parameters set (if necessary). The passed parameters and their order must correspond with the declaration order and the type of extern variables of the custom indicator.
mode	- Line index. Can be from 0 to 7 and must correspond with the index used by one of SetIndexBuffer functions.
shift	- Index of the value taken from the indicator buffer (shift relative to the current bar the given amount of periods ago).

Sample:

```
double val=iCustom(NULL, 0, "SampleInd",13,1,0);
```

What makes the *iCustom* function difficult to use is the fourth parameter. The ellipsis (...) indicates there can be zero to N number of parameters defined. The actual number depends on the specific custom indicator.

The mode parameter is used to get the value of the specific line output of the custom indicator. Many indicators have more than one line drawn on the chart. This value must be defined correctly to receive the value of the correct line.

The remaining parameters are straightforward.

Note: An indicator does not need to be drawn on a price chart to be used by an Expert Advisor.