

---

# Table of Contents

---

<b>Introduction</b>	<b>1</b>
About This Book	2
A Note About MQL 5	2
Conventions Used In This Book	3
<b>An Introduction to MQL</b>	<b>4</b>
Introduction to MetaEditor	4
Basic Concepts	7
Layout of an MQ4 File	14
<b>Order Placement</b>	<b>20</b>
Bid, Ask & Spread	20
Order Types	20
The Order Placement Process	21
OrderSend()	22
Calculating Stop Loss & Take Profit	25
Retrieving Order Information	32
Closing Orders	34
A Simple Expert Advisor	36
<b>Advanced Order Placement</b>	<b>42</b>
Order Modification	42
Verifying Stops and Pending Order Prices	45
Calculating Lot Size	49
Other Considerations	52
Putting It All Together	56
<b>Working with Functions</b>	<b>64</b>
Add Stop Loss and Take Profit	73
Using Include Files	74

Using Libraries	74
A Simple Expert Advisor (with Functions)	75
<b>Order Management</b>	<b>80</b>
The Order Loop	80
Order Counting	82
Trailing Stops	87
Updating the Expert Advisor	91
<b>Order Conditions and Indicators</b>	<b>94</b>
Price Data	94
Indicators	95
Indicator Constants	102
Evaluating Trade Conditions	103
Comparing Indicator Values Across Bars	108
<b>Working with Time and Date</b>	<b>112</b>
Datetime Variables	112
Date and Time Functions	114
Creating A Simple Timer	115
Execute On Bar Open	117
<b>Tips and Tricks</b>	<b>122</b>
Escape Characters	122
Using Chart Comments	122
Check Settings	123
Demo or Account Limitations	124
MessageBox()	125
Email Alerts	127
Retry on Error	128
Using Order Comments As an Identifier	131
Margin Check	132
Spread Check	133
Multiple Orders	133

Global Variables	136
Check Order Profit	138
Martingale	138
Debugging Your Expert Advisor	141
<b>Custom Indicators and Scripts</b>	<b>146</b>
Buffers	146
Creating A Custom Indicator	146
Scripts	152
<b>Appendix A</b>	<b>154</b>
Simple Expert Advisor	154
Simple Expert Advisor with Pending Orders	156
<b>Appendix B</b>	<b>160</b>
Advanced Expert Advisor	160
Advanced Expert Advisor with Pending Orders	166
<b>Appendix C</b>	<b>172</b>
Advanced Expert Advisor with Functions	172
Advanced Expert Advisor with Functions – Pending Orders	175
<b>Appendix D</b>	<b>180</b>
Include File	180
<b>Appendix E</b>	<b>198</b>
Custom Indicator	198



---

# Introduction

---

The foreign exchange market has rapidly become one of the most popular markets to trade in recent years. Because of its round-the-clock hours, high leverage and low margin requirements, thousands of ordinary people have become active traders.

MetaTrader 4 (commonly abbreviated as MT4) has become one of the most popular trading platforms for forex. Developed by MetaQuotes Software Corporation, MetaTrader is offered by hundreds of forex brokers worldwide, including big names such as GAIN Capital, FXCM, Alpari and Interbank FX.

MetaTrader's popularity stems from the fact that it's free, broker supported, and includes many useful technical analysis tools. But probably the biggest reason for MetaTrader's success is the powerful MQL programming language.

MQL has made it possible for traders to program their own custom indicators and automated trading strategies without paying a dime for software. Similar trading packages for equities and futures can cost over \$1000. A worldwide community of traders and programmers has developed, offering hundreds of free and commercial expert advisors and indicators, as well as programming services and advice.

The similarity of MQL to languages such as C makes it relatively easy for experienced programmers to pick up, and the language itself is well documented. But learning how to effectively program trading strategies in MQL is a process of trial and error.

MQL is a relatively low level language, and as such, it is necessary for the programmer to create custom procedures to handle many common trading functions. Coding something as simple as a trailing stop, for example, can be daunting for the new MQL programmer.

There are many factors that must be taken into consideration when programming a robust automated trading strategy, and MetaTrader itself has many idiosyncrasies that the programmer needs to be aware of. It can take dozens of hours of troubleshooting and practice to learn the techniques necessary to program expert advisors.

This book hopes to shorten the learning curve for new expert advisor programmers. Here I will present many of the tips and tricks I've learned in the hundreds of hours I've spent coding expert advisors over the last few years.

## About This Book

By the time you finish this book, you should possess the knowledge necessary to create your own robust automated trading strategies in MQL, including common trading features such as trailing stops, money management and much more. You will also learn how to construct a simple indicator, using built-in indicator functions.

This book assumes that the reader is knowledgeable about forex trading and technical analysis in general. The reader should already be proficient in using expert advisors and indicators in MetaTrader. While no prior programming knowledge is assumed, the reader will benefit from having some basic programming skills, and familiarity with concepts such as variables, control structures, functions and modern programming language syntax.

We will be diving right into coding solutions to specific problems. Every attempt is made to explain new concepts as they are introduced, however this book is not intended as a language reference. The MQL reference at <http://docs.mql4.com> does an excellent job at that. The MQL reference is also built into the MetaEditor IDE that comes with MetaTrader.

While we will attempt to touch on everything that is necessary and relevant to expert advisor development, we will not be able to cover every element of the MQL language. There are many specialized functions in MQL that are not generally used in expert advisor programming. In particular, we will not be discussing array functions, file manipulation, objects, windows, and most string or conversion functions.

The official MQL4 website at <http://www.mql4.com> has a free book on MQL programming that may serve as a useful and complementary resource. There are many informative articles that cover basic and advanced programming concepts in MQL, a code library with additional indicators and examples, and a forum where you can ask for help with your programming questions.

The code examples and techniques I teach in this book are what has worked for me. I try to keep things as simple as possible, without sacrificing functionality. That said, there is always more than one way to accomplish something, and this is especially true in programming. There are equally valid methods of achieving the same result, and it is possible you may discover a better way of doing something.

## A Note About MQL 5

As of this writing, the next version of the MetaTrader platform is in open beta testing. MetaTrader 5 is expected to be released sometime in 2010. There will be some significant changes to the newest version

of MQL. MetaQuotes has reported that MetaTrader 5 will not be backward compatible with MetaTrader 4 programs. Thus, any programs written in MQL 4 will need to be rewritten or updated for MQL 5.

This book deals with MetaTrader 4, as it is the version I have been programming in for the last few years and is currently the version that is being used by Forex brokers. Since the release of MetaTrader 4 in 2005, Forex trading has exploded in popularity. MetaTrader has become the most popular forex trading platform, and there have been thousands of trading strategies and indicators written in MQL 4.

I predict the migration to MetaTrader 5 will be a gradual one. Brokers will continue to support MetaTrader 4 for some time, so the programs you write in MQL 4 will not become obsolete immediately. The concepts in this book will remain the same, although some of the functions and syntax will change. The challenge will be to learn the new MQL 5 features and incorporate it into your existing code.

Once MetaTrader 5 has been released, an addendum to this book will be written, addressing the changes between MQL 5 and the previous version and updating much of the code. It will be available as a free download at <http://www.expertadvisorbook.com>. I cannot currently predict a time frame for release, but expect to see it several months after the commercial release of MetaTrader 5.

## Conventions Used In This Book

MQL language elements, source code examples, and file and URL locations will be displayed in a **fixed-width font**. A larger bold font will be used for inline text. Blocks of source code will be indented. Any bold text appearing in an indented source code block indicates code that has been updated or changed from a previous example.

Source code block  
**Updated source code**

Words in *italics* indicate a new concept that is being introduced or defined. References to sections and topics in the MQL Reference will be displayed in italics. References to elements of the MetaTrader 4 interface, including windows, dialogs, buttons or menu items, will also be displayed in italics.

---

# Chapter 1

## *An Introduction to MQL*

---

### Introduction to MetaEditor

#### What is an Expert Advisor?

An *expert advisor* is an automated trading program written in MQL. Expert advisors (commonly abbreviated as EA) can place, modify and close orders according to a trading system algorithm. EA's generally use indicators to generate trading signals. These indicators can be the ones that come with MetaTrader, or they can be custom indicators.

An *indicator* is a technical analysis tool that calculates price data to give an interpretation of market activity. An indicator draws lines or objects on the chart. Indicators cannot place, modify or close orders. Examples of indicators include the moving average and stochastics.

A *script* is a simplified expert advisor that performs a single task, such as placing a pending order or closing all orders on a chart. A few useful scripts are included with MetaTrader.

#### File Formats

Files with the `.mq4` extension are *source code* files. These are the files we edit in MetaEditor. When an `.mq4` file is compiled, an `.ex4` file is produced.

Files with the `.ex4` extension are *executable* files. These are the files we run in MetaTrader. These files cannot be opened in MetaEditor. If you only have the `.ex4` file for an EA or indicator, the icon next to the file name in MetaTrader's *Navigator* window will be grayed out.

Files with the `.mqh` extension are *include* files. These files contain user-created functions that are referenced in an `.mq4` file. During compilation, the compiler "includes" the contents of the `.mqh` file in the `.ex4` file. We'll learn more about include files later.

The `.mq5` extension is used for template files. While these files can be opened in MetaTrader, the file type is not associated with the program in Windows. Templates are used to create new files using the Expert Advisor Wizard in MetaEditor.

You can create your own templates if you wish, but we will not be covering template creation in this book. The MetaTrader documentation will tell you all you need to know about creating templates.

Indicators, expert advisors, libraries and scripts all share the **.mq4** extension. The only way to tell them apart is either by their save location, or by opening the file and examining them. By the time you finish this book, you should be able to identify the difference between program types just by looking at the source code.

## File Locations

All MetaEditor files are stored inside the *experts folder*. The **\experts** folder is contained in the MetaTrader installation directory, which is in **C:\Program Files\**. If your broker is Interbank FX, for example, the MT4 installation folder would be **C:\Program Files\Interbank FX Trader 4\**.

The **\experts** folder contains the source code and executable files for the expert advisors. Using the above example, the **\experts** folder would be located at **C:\Program Files\Interbank FX Trader 4\experts\**.

There are numerous folders inside the **\experts** folder that contain other types of source code and executable files. Here's a list of the save locations for all file types:

- **\experts\indicators** – Source code and executable files for your indicators are stored here.
- **\experts\include** – Source code include files with the **.mqh** extension are stored here.
- **\experts\libraries** – Function libraries and DLLs are saved here.
- **\experts\scripts** – Source code and executable files for scripts are stored here.
- **\experts\templates** – Templates for source code files are stored here.

There are a few other folders inside the experts folder that you'll want to be aware of too:

- **\experts\logs** – Activity logs for your expert advisors are stored here. These will be useful for debugging your expert advisors.
- **\experts\presets** – Expert advisor settings that are saved or loaded from MetaTrader's *Properties* dialog are stored here.
- **\experts\files** – Any files used for input or output must be stored here.

## MetaEditor

MetaEditor is an Integrated Development Environment (IDE) for MQL that comes packaged with MetaTrader. It includes useful reference, search and auto-complete tools that makes coding in MQL a lot easier.

The *Editor* window allows you to have multiple files open at once. You can minimize, maximize and tab between several open windows. The *Navigator* window offers useful file-browsing and reference features. The *Toolbox* window displays help contents, compilation errors, file search results, and online access to articles and files at MQL4.com.



Fig. 1.1 – The MetaEditor interface. Clockwise from top left: Editor window, Navigator window, and Toolbox window.

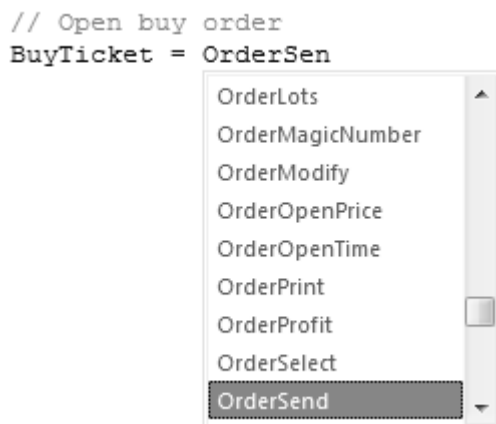
One of the most useful editing features is the Assistant. Simply type the first few characters of an MQL function, operator or other language element, and a drop-down list will appear. Press Enter to accept the highlighted suggestion and auto-complete the phrase.

The *Files* tab in the Navigator window is a simple file browser that allows you to open and edit any of the MQL files in your `\experts` folder. The *Dictionary* tab features a built-in MQL reference, while the *Search* tab is a search feature for the MQL reference.

The built-in MQL reference and the context-sensitive help will save you a lot of time when coding. If you need help remembering the syntax of a particular language element, select or place the text cursor on the element in the editor window. Press F1 on your keyboard and the help topic will appear in the Toolbox window.

The toolbar in MetaEditor features the standard complement of file and editing functions. The Navigator and Toolbox windows can be shown or hidden using their respective buttons on the toolbar.

The *Compile* button compiles the current file in the editor. If there are any compilation errors, they will be shown in the Toolbox window. The *Terminal* button opens the trading terminal for testing.



**Fig. 1.2** – MetaEditor's Assistant auto complete feature.

## Basic Concepts

We're going to review some basic programming concepts that will make the rest of this book easier to understand for new programmers. If you're an experienced programmer, feel free to skip ahead to the next section, *Layout of an MQL File*.

### Syntax

If you're familiar with programming in languages such as C++, PHP or one of the many languages whose syntax is derived from C, you'll be very comfortable programming in MQL. If your previous programming experience is in a language such as Visual Basic, then you may need to make a few adjustments.

In MQL, every statement is terminated with a semicolon. This is called an *expression*. An expression can span multiple lines, but there must be a semicolon at the end.

```
double LastHigh = High[1];

string MultiLine = StringConcatenate("This is a multi-line statement. ",
    "For clarity, we will indent multiple lines in this book");
```

If you're new to programming, or accustomed to programming in a language that does not terminate expressions with a semicolon, you'll need to make sure you're placing the semicolon at the end of every statement. Not terminating lines with a semicolon is a common newbie mistake.

There are a few exceptions to this: Compound operators do not need a semi-colon. A *compound operator* is a block of code that contains multiple expressions within braces {}. Examples of compound operators include control operators (**if**, **switch**), cycle operators (**for**, **while**) and function declarations.

```
if(Compound == true)
{
    Print("This is a compound expression");
}
```

Note that there is no semicolon after the initial **if** operator, nor is there a semicolon after the closing brace. There is a semicolon after the **Print()** function, however. There can be one, or multiple expressions inside the braces. Each must end with a semicolon.

## Comments

Comments are useful for documenting your code, as well as for temporarily removing code while testing and debugging. You can comment out a single line with two forward slashes:

```
// This is a comment
```

A multi-line comment begins with `/*` and ends with `*/`. A multi-line comment can span any number of lines, and everything between `/*` and `*/` is commented out.

```
/* This is a comment block
   Everything here is commented out */
```

## Identifiers

Identifiers are names given to variables and custom functions. An identifier can be any combination of numbers, letters, and the underscore character (`_`). Identifiers can be up to 31 characters in length.

You'll want your identifiers to be descriptive of their function, but be sure your identifier doesn't match an MQL language element (also called a *reserved word*). Here's an example of a variable identifier and a custom function identifier. The identifier is in italics:

```
double StopLoss;
int Order_Count()
```

Identifiers in MQL are *case-sensitive*. This means that **StopLoss** and **stoploss** are different variables! This is another common newbie mistake, so check those identifier names!

## Variables

A *variable* is the basic storage unit of any programming language. Variables hold data necessary for our program to function, such as prices, settings and indicator values.

Variables must be declared before they are used. To declare a variable, you specify its *data type*, an identifier, and optionally a default value. If you declare a variable more than once, or not at all, you'll get a compilation error.

The *data type* specifies the type of information the variable holds, whether it be a number, a text string, a date or a color. Here are the data types in MQL:

- **int** – A integer (whole number) such as 0, 3, or -5. Any number assigned to an integer variable is rounded up to the next whole number.
- **double** – A fractional number such as 1.5765, 0.03 or -2.376. Use these for price data, or in mathematical expressions involving division.
- **string** – A text string such as "The quick brown fox jumped over the lazy dog". Strings must be surrounded by double quotes.
- **boolean** – A **true/false** value. Can also be represented as 1 (true) or 0 (false). Use these anytime you need to evaluate an binary, or on/off condition.
- **datetime** – A time and date value such as **2009.01.01 00:00**. Internally, a datetime variable is represented as the number of seconds passed since January 1, 1970.
- **color** – A constant representing a color, such as **Red** or **DarkSlateBlue**. These are generally used for changing indicator or object colors.

Here's an example of a variable declaration. This is an integer variable, with the identifier **MyVariable** and a default value of 1.

```
int MyVariable = 1;
```

Once a variable has been declared, you can change its value by assigning a new value to it. Here's an example where we assign the number 5 to **MyVariable**:

```
MyVariable = 5;
```

You can also assign the value of one variable to another variable:

```
int YourVariable = 2;
MyVariable = YourVariable;
// MyVariable is 2
```

The assigned variable should be of the same data type. If a double is assigned to an integer variable, for example, the double will be rounded to the nearest whole number. This may lead to an undesirable result.

## Constants

Just like its name suggests, a *constant* is a data value that never changes. For example, the number 5 is an integer constant, the letter 'A' is a character constant, and **2009.01.01** is a datetime constant for January 1, 2009.

MQL has a wide variety of standard constants for things like price data, chart periods, colors and trade operations. For example **PERIOD\_H1** is a constant for the H1 chart time frame, **OP\_BUY** refers to a buy market order, and **Red** is a color constant for the color red.

You can even create your own constants using the **#define** preprocessor directive. We'll get to that shortly. You can learn more about MQL's standard constants in the *Standard Constants* section of the MQL Reference.

## Functions

Functions are the building blocks of modern programming languages. A function is a block of code that is designed to carry out a certain task, such as placing an order or calculating a stop loss. MQL has dozens of built-in functions for everything from technical indicators to order placement.

Functions are designed to be reused over and over again. Learning how to create functions for common trading tasks is essential to productive programming. We will work on creating reusable functions for many of the tasks that we will learn in this book.

Let's start with a simple function called **PipPoint()**, that calculates the number of decimal points in the current pair, and automatically adjusts for 3 and 5 digit brokers so that the result is always equal to one pip. For Yen pairs (2 or 3 digits), the function returns 0.01. For all other pairs (4 and 5 digits), the function returns 0.0001. Here's how we would call the function from code:

```
double UsePoint;
UsePoint = PipPoint();
```

We declare a variable of type **double** named **UsePoint**. Then we call the **PipPoint()** function and assign the result to **UsePoint**. Now we can use the value stored in **UsePoint** to calculate a stop loss, for example.

Here is the code for the **PipPoint()** function:

```
double PipPoint()
{
    if(Digits == 2 || Digits == 3) double UsePoint = 0.01;
    else if(Digits == 4 || Digits == 5) UsePoint = 0.0001;
    return(UsePoint);
}
```

The first line is our function declaration. Like variables, function declarations have a data type and an identifier. Functions use the same data types as variables do. The data type is dependent on the type of data the function returns. Since this function returns a fractional number, we use the **double** data type.

The body of the function is contained within the brackets **{}**. We have an **if-else** statement that evaluates the number of digits after the decimal place, and assigns the appropriate value to the **UsePoint** variable. Following that, we have the **return** operator, which returns the value of **UsePoint** to the calling function.

There is a special data type for functions that do not return a value. The **void** data type is used for functions that carry out a specific task, but do not need to return a value to the calling function. **Void** functions do not require a **return** operator in the body of the function.

Let's consider a simple function for placing a buy order. This function has *arguments* that need to be passed to the function. This function will place a buy market order on the current symbol with the specified lot size, stop loss and take profit.

```
int OpenBuyOrder(double LotSize, double StopLoss, double TakeProfit)
{
    int Ticket = OrderSend(Symbol(), OP_BUY, LotSize, Ask, StopLoss, TakeProfit);
    return(Ticket);
}
```

This function has three arguments, **LotSize**, **StopLoss** and **TakeProfit**. Arguments are variables that are used only within the function. Their value is assigned by the calling function. Here's how we would call this function in code using constants:

```
OpenBuyOrder(2, 1.5550, 1.6050);
```

This will place a buy order of 2 lots, with a stop loss of 1.5550 and a take profit of 1.6050. Here's another example using variables. We'll assume that the variables **UseLotSize**, **BuyStopLoss** and **BuyTakeProfit** have the appropriate values assigned:

```
int GetTicket = OpenBuyOrder(UseLotSize,BuyStopLoss,BuyTakeProfit);
```

In this example, we are assigning the return value of **OpenBuyOrder()** to the variable **GetTicket**, which is the ticket number of the order we just placed. Assigning the output of a function to a variable is optional. In this case, it is only necessary if you plan to do further processing using the ticket number of the placed order.

Arguments can have *default values*, which means that if a parameter is not explicitly passed to the function, the argument will take the default value. Default value arguments will always be at the end of the argument list. Here is an example of a function with several default values:

```
int DefaultValFunc(int Ticket, double Price, int Number = 0, string Comment = NULL)
```

This function has two arguments with default values, **Number** and **Comment**, with default values of **0** and **NULL** respectively. If we want to use the default values for both **Number** and **Comment**, we simply omit those arguments when calling the function:

```
DefaultValFunc(TicketNum,UsePrice);
```

Note that we only specified the first two arguments. **Number** and **Comment** use the default values of **0** and **NULL**. If we want to specify a value for **Number**, but not for **Comment**, we simply omit the last argument:

```
DefaultValFunc(TicketNum,UsePrice,UseNumber);
```

Again, **Comment** uses the default value of **NULL**. But, if we want to specify a value for **Comment**, regardless of whether or not we want to use the default value for **Number**, we have to specify a value for **Number** as well:

```
DefaultValFunc(TicketNum,UsePrice,0,"Comment String");
```

In this example, we used 0 as the value for **Number**, which is the same as the default value, and a string constant as the value for **Comment**. Remember that when you're dealing with multiple arguments that have default values, you can only omit arguments if you want to use the default values for all of the remaining arguments!